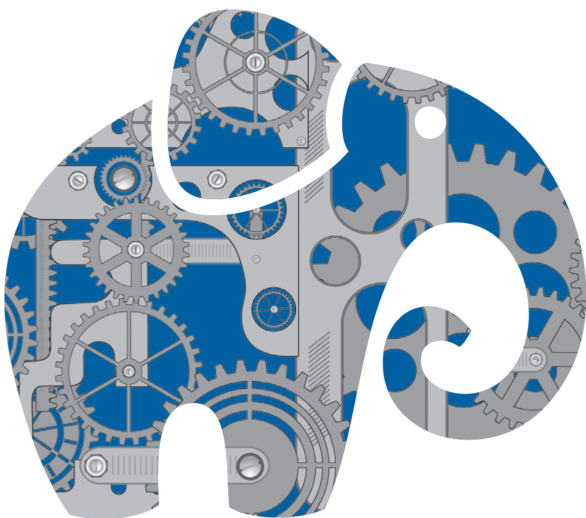
郑钢 ◉ 著

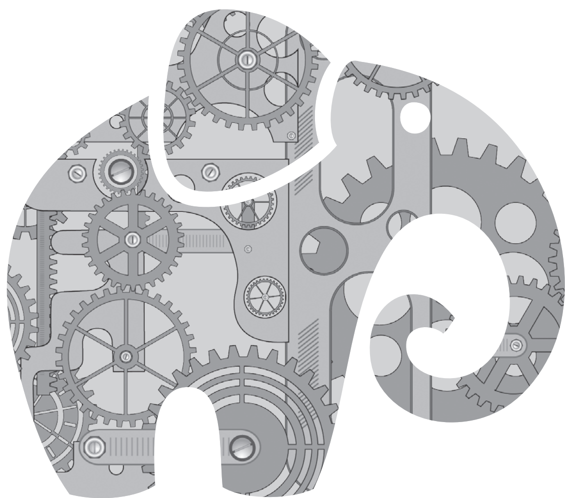
历时 19 个月，行文 60 余万字，
用 6000 多行代码实现了一个完整的操作系统。
彻底剖析操作系统的原理，实现内核线程、特权级变换、
用户进程、任务调度、文件系统等操作系统最基本的组成单元。
用实际代码解释了锁、信号量、生产者、消费者问题。
实现了一个简单的 shell，帮助大家理解内部命令、
外部命令、管道等操作。
操作系统并不深奥，本书给予权威解读。



操作系统真象还原

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

郑钢 著



操作系统真象还原

人民邮电出版社
北京

异步社区
人民邮电出版社
www.epubit.com.cn

异步社区电子书

感谢您购买异步社区电子书！异步社区已上架电子书 500 余种，社区还会经常发布福利信息，对社区有贡献的读者赠送免费样书券、优惠码、积分等等，希望您在阅读过程中，把您的阅读体验传递给我们，让我们了解读者心声，有问题我们会及时修正。

社区网址： http://www.epubit.com.cn/

反馈邮箱： contact@epubit.com.cn

异步社区里有什么？

图书、电子书（半价电子书）、优秀作译者、访谈、技术会议播报、赠书活动、下载资源。

异步社区特色：

纸书、电子书同步上架、纸电捆绑超值优惠购买。

最新精品技术图书全网首发预售。

晒单有意外惊喜！

异步社区里可以做什么？

博客式写作发表文章，提交勘误赚取积分，积分兑换样书，写书评赢样书券等。

联系我们：

微博：

@人邮异步社区　　@人民邮电出版社 - 信息技术分社

微信公众号：

人邮 IT 书坊　　异步社区

QQ 群： 368449889

图书在版编目（ＣＩＰ）数据

操作系统真象还原 / 郑钢著. -- 北京 : 人民邮电出版社, 2016.3
ISBN 978-7-115-41434-2

Ⅰ. ①操… Ⅱ. ①郑… Ⅲ. ①操作系统－基本知识 Ⅳ. ①TP316

中国版本图书馆CIP数据核字(2016)第016739号

内 容 提 要

本书共分 16 章，讲解了开发一个操作系统需要的技术和知识，主要内容有：操作系统基础、部署工作环境、编写 MBR 主引导记录、完善 MBR 错误、保护模式入门、保护模式进阶和向内核迈进、中断、内存管理系统、线程、输入输出系统、用户进程、完善内核、编写硬盘驱动程序、文件系统、系统交互等核心技术。

本书适合程序员、系统底层开发人员、操作系统爱好者阅读，也可作为大专院校相关专业师生用书和培训学校的教材。

◆ 著 郑 钢
责任编辑 张 涛
责任印制 张佳莹 焦志炜

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
三河市中晟雅豪印务有限公司印刷

◆ 开本：787×1092 1/16
印张：48.25
字数：1 281 千字 2016 年 3 月第 1 版
印数：1 – 2 500 册 2016 年 3 月河北第 1 次印刷

定价：108.00 元

读者服务热线：(010)81055410 印装质量热线：(010)81055316
反盗版热线：(010)81055315

前　　言

本书面向操作系统基础知识薄弱，但又想把操作系统搞清楚、喜欢刨根问底的技术人，在此向你们致敬，本书用诙谐幽默的语言，把深奥的操作系统尽量讲解清楚，读者在轻松阅读中就学通了深奥的知识，是一本难得的好书。

多数学习操作系统的读者都会有这样的感受：

（1）“太难了，对于操作系统这个庞然大物我简直无从下手”；

（2）“很后悔选了这门课（大学一些专业中操作系统是选修课），甚至不想学习计算机了”；

（3）“上课完全听不懂，我都不想继续听下去了”；

（4）“即使实验做出来了，由于只是完成了局部功能，我依然不明白操作系统是怎样运行起来的，甚至不知道自己在做什么”。

以上的感受我都有过，坦白说，这门课并不是很难，但想把这门课完全搞明白真不容易。我是个喜欢刨根问底的人，为了弄清楚这背后的真相，我花了大量时间学习课程之外的内容，甚至付出了惨痛的代价——大学中第一次考试不及格，操作系统这门课我是第二次才考过的。这确实很“讽刺”——操作系统不及格的人在写操作系统书籍！但转念一想，考试过了的同学并不代表能够写出操作系统，因为试卷上并不是在考如何写一个操作系统。和技术能力相比，卷面成绩并不重要。

想象一下，如果是爱因斯坦那样的天才给我们讲物理知识，我们会觉得物理更容易理解吗？肯定是不会的，因为在爱因斯坦眼中比较容易的内容也许对我们来说非常深奥，他用 B 解释 A 的时候也许会让我们更迷惑，因为 B 我们也不懂，这就是基础的问题了。幸运的是阅读本书时读者只要有 C 语言和部分汇编语言的基础就行了，涉及的其他方面的知识我都会详细介绍，并以更易懂的方式去解释技术难点，读者不必担心看不懂本书。

回忆一下学车的经历：教练让学员先踩离合器再挂档，然后再踩油门，车子就开动啦。如果学员总是学不会这些，有可能是学员根本不知道什么是离合器，或者不知道离合器的作用是什么。即使把这些操作背下来，也会对驾车感到心有余而力不足，可见，只有了解了背后的原理，才会知道自己在做什么，驾车才变得游刃有余。

以上情况对我们学习操作系统来说也同样存在，比如当老师介绍中断发生时的上下文保护时，我们更多的疑问不是如何保存 CPU 的上下文数据，而是想知道为什么在不同的特权级下会使用不同的栈，这背后的原理是什么，并且这是如何做到的。

诸如此类的疑问需要了解硬件原生支持的运行机制，因为很多操作都是硬件自动完成的，比如处理器进入 0 特权级时，会自动在任务状态段 TSS 中获得 0 特权级的栈地址，这不需要人工干涉，完全由处理器维护。我们想知道的是，硬件在背后自动完成了哪些工作，这样才便于我们理解操作系统的全貌。

操作系统受制于硬件的支持，很大程度上它的能力取决于硬件的能力，因此，要想全面理解操作系统，不仅需要了解上层软件的算法、原理、实现，还要了解很多硬件底层的内容。和硬件相关的知识是在微机接口电路中讲解的，而绝大多数读者在学习这门课时，根本不知道它有何用，只有学习操作系统课程时才用到它，因此，本书内容兼顾相关的硬件知识。

除硬件外，本书还把操作系统中的理论付诸于实践，让读者真正学到包含在操作系统中的实实在在的技术，比如在代码中实现了著名的生产者消费者问题，还有进程、线性、阻塞、信号量、锁、文件系统、目录、shell、管道等。各个章节的代码都可独立运行，方便调试，本书更让读者有成就感的是，我们最终完成的一个操作系统总共代码量只有几千行左右，极大地减少了操作系统源码阅读的工作量。

操作系统还是比较庞大的，因此，大部分介绍操作系统原理的书中，对各个部分都是分拆出来介绍的，这导致我们学习操作系统时犹如盲人摸象、管中窥豹。本书的封面是一个完整的大象的拼图，就像封面展示的那样，本书内容我们不再局部学习，而是把所有局部还原成一个整体，做出一个真正的操作系统。

为了让读者不再惧怕操作系统，同时也为了完成我自己的心愿，我辞职专心进行本书的编写，在此期间也曾拒绝了多份回报丰厚的工作，现在想想真是疯狂……苦了我的父母和女朋友，在这里跟你们说声抱歉，你们“纵容”我的偏执，真心不容易，辛苦啦，我爱你们！

感谢我在北京大学就读期间的 Linux 内核课程老师（同时也是我的研究生导师）荆琦教授和操作系统课程老师陈向群教授，很荣幸能够成为您们的学生，时至今日我常常回想起课堂上您们言传身教并为我解答问题的身影，您们渊博的知识和教学上严谨的态度深深影响了我，仅以此书向我这两位恩师致谢。

感谢父母给予我的理解和宽容，以后我一定加倍努力回报您们的养育之恩！

最后，感谢女朋友给予我的陪伴和照顾，在写此书的过程中我深深体会到：爱并不仅仅体现在相信对方一定能成功，更多是体现在支持对方去做想做的事，即使失败了也不会嫌弃。尽管在这漫长枯燥的 19 个月当中，如果没有你的“唠唠叨叨”本书早就写完了，但恰恰是这种“唠唠叨叨”下的不离不弃让我相信这世上还有真爱。

我爱你王小兔（对我女朋友的昵称），本书是我送给你的礼物。

本书中出现的“兄弟”“大伙儿”“同学”和“咱们”的称谓，是作者为了活泼写作风格故意为之，别无他意，在此说明一下。本书读者交流 QQ 群为：148177180，编辑联系邮箱：zhangtao@ptpress.com.cn。

作　者
于北京大学图书馆

目　录

第 0 章　一些你可能正感到迷惑的问题……1

0.1　操作系统是什么……1
0.2　你想研究到什么程度……2
0.3　写操作系统，哪些需要我来做……2
0.4　软件是如何访问硬件的……2
0.5　应用程序是什么，和操作系统是如何配合到一起的……3
0.6　为什么称为“陷入”内核……4
0.7　内存访问为什么要分段……4
0.8　代码中为什么分为代码段、数据段？这和内存访问机制中的段是一回事吗……6
0.9　物理地址、逻辑地址、有效地址、线性地址、虚拟地址的区别……11
0.10　什么是段重叠……12
0.11　什么是平坦模型……12
0.12　cs、ds 这类 sreg 段寄存器，位宽是多少……12
0.13　什么是工程，什么是协议……13
0.14　为什么 Linux 系统下的应用程序不能在 Windows 系统下运行……14
0.15　局部变量和函数参数为什么要放在栈中……14
0.16　为什么说汇编语言比 C 语言快……15
0.17　先有的语言，还是先有的编译器，第 1 个编译器是怎么产生的……16
0.18　编译型程序与解释型程序的区别……19
0.19　什么是大端字节序、小端字节序……19
0.20　BIOS 中断、DOS 中断、Linux 中断的区别……21
0.21　Section 和 Segment 的区别……25
0.22　什么是魔数……29
0.23　操作系统是如何识别文件系统的……30
0.24　如何控制 CPU 的下一条指令……30
0.25　指令集、体系结构、微架构、编程语言……30
0.26　库函数是用户进程与内核的桥梁……33
0.27　转义字符与 ASCII 码……37
0.28　MBR、EBR、DBR 和 OBR 各是什么……39

第 1 章　部署工作环境……42

1.1　工欲善其事，必先利其器……42
1.2　我们需要哪些编译器……42
1.2.1　世界顶级编译器 GCC……42
1.2.2　汇编语言编译器新贵 NASM……43
1.3　操作系统的宿主环境……43
1.3.1　什么是虚拟机……44
1.3.2　盗梦空间般的开发环境，虚拟机中再装一个虚拟机……45
1.3.3　virtualBox 下载，安装……46
1.3.4　Linux 发行版下载……46
1.3.5　Bochs 下载安装……46
1.4　配置 bochs……48
1.5　运行 bochs……49

第 2 章　编写 **MBR** 主引导记录，让我们开始掌权……52

2.1　计算机的启动过程……52
2.2　软件接力第一棒，BIOS……52
2.2.1　实模式下的 1MB 内存布局……52
2.2.2　BIOS 是如何苏醒的……54
2.2.3　为什么是 0x7c00……56
2.3　让 MBR 先飞一会儿……58
2.3.1　神奇好用的$和$$，令人迷惑的 section……58
2.3.2　NASM 简单用法……60
2.3.3　请下一位选手 MBR 同学做准备……60

第 3 章　完善 **MBR**……65

3.1　地址、section、vstart 浅尝辄止……65
3.1.1　什么是地址……65
3.1.2　什么是 section……67
3.1.3　什么是 vstart……68
3.2　CPU 的实模式……70
3.2.1　CPU 的工作原理……71
3.2.2　实模式下的寄存器……72
3.2.3　实模式下内存分段由来……76
3.2.4　实模式下 CPU 内存寻址方式……78
3.2.5　栈到底是什么玩意儿……81

3.2.6 实模式下的 ret……84
3.2.7 实模式下的 call……85
3.2.8 实模式下的 jmp……92
3.2.9 标志寄存器 flags……97
3.2.10 有条件转移……99
3.2.11 实模式小结……101
3.3 让我们直接对显示器说点什么吧……101
3.3.1 CPU 如何与外设通信——IO 接口……101
3.3.2 显卡概述……105
3.3.3 显存、显卡、显示器……106
3.3.4 改进 MBR，直接操作显卡……110
3.4 bochs 调试方法……112
3.4.1 bochs 一般用法……113
3.4.2 bochs 调试实例……118
3.5 硬盘介绍……122
3.5.1 硬盘发展简史……122
3.5.2 硬盘工作原理……123
3.5.3 硬盘控制器端口……126
3.5.4 常用的硬盘操作方法……128
3.6 让 MBR 使用硬盘……129
3.6.1 改造 MBR……130
3.6.2 实现内核加载器……134

第 4 章 保护模式入门……136

4.1 保护模式概述……136
4.1.1 为什么要有保护模式……136
4.1.2 实模式不是 32 位 CPU，变成了 16 位……137
4.2 初见保护模式……137
4.2.1 保护模式之寄存器扩展……137
4.2.2 保护模式之寻址扩展……140
4.2.3 保护模式之运行模式反转……141
4.2.4 保护模式之指令扩展……145
4.3 全局描述符表……150
4.3.1 段描述符……150
4.3.2 全局描述符表 GDT、局部描述符表 LDT 及选择子……155
4.3.3 打开 A20 地址线……157
4.3.4 保护模式的开关，CR0 寄存器的 PE 位……158
4.3.5 让我们进入保护模式……158
4.4 处理器微架构简介……165
4.4.1 流水线……166
4.4.2 乱序执行……168
4.4.3 缓存……168
4.4.4 分支预测……169
4.5 使用远跳转指令清空流水线，更新段描述符缓冲寄存器……172
4.6 保护模式之内存段的保护……173
4.6.1 向段寄存器加载选择子时的保护……173
4.6.2 代码段和数据段的保护……174
4.6.3 栈段的保护……175

第 5 章 保护模式进阶，向内核迈进……177

5.1 获取物理内存容量……177
5.1.1 学习 Linux 获取内存的方法……177
5.1.2 利用 BIOS 中断 0x15 子功能 0xe820 获取内存……177
5.1.3 利用 BIOS 中断 0x15 子功能 0xe801 获取内存……179
5.1.4 利用 BIOS 中断 0x15 子功能 0x88 获取内存……180
5.1.5 实战内存容量检测……181
5.2 启用内存分页机制，畅游虚拟空间……186
5.2.1 内存为什么要分页……186
5.2.2 一级页表……188
5.2.3 二级页表……192
5.2.4 规划页表之操作系统与用户进程的关系……197
5.2.5 启用分页机制……198
5.2.6 用虚拟地址访问页表……204
5.2.7 快表 TLB（Translation Lookaside Buffer）简介……206
5.3 加载内核……207
5.3.1 用 C 语言写内核……207
5.3.2 二进制程序的运行方法……211
5.3.3 elf 格式的二进制文件……213
5.3.4 elf 文件实例分析……218
5.3.5 将内核载入内存……222
5.4 特权级深入浅出……229
5.4.1 特权级那点事……229
5.4.2 TSS 简介……230
5.4.3 CPL 和 DPL 入门……232
5.4.4 门、调用门与 RPL 序……235
5.4.5 调用门的过程保护……240
5.4.6 RPL 的前世今生……243
5.4.7 IO 特权级……248

第 6 章 完善内核……252

6.1 函数调用约定简介……252

6.2 汇编语言和 C 语言混合编程 ······256
6.2.1 浅析 C 库函数与系统调用 ······256
6.2.2 汇编语言和 C 语言共同协作 ······259
6.3 实现自己的打印函数 ······261
6.3.1 显卡的端口控制 ······261
6.3.2 实现单个字符打印 ······265
6.3.3 实现字符串打印 ······275
6.3.4 实现整数打印 ······277
6.4 内联汇编 ······281
6.4.1 什么是内联汇编 ······281
6.4.2 汇编语言 AT&T 语法简介 ······281
6.4.3 基本内联汇编 ······283
6.4.4 扩展内联汇编 ······284
6.4.5 扩展内联汇编之机器模式简介 ······294

第 7 章 中断 ······298

7.1 中断是什么，为什么要有中断 ······298
7.2 操作系统是中断驱动的 ······299
7.3 中断分类 ······299
7.3.1 外部中断 ······299
7.3.2 内部中断 ······301
7.4 中断描述符表 ······304
7.4.1 中断处理过程及保护 ······306
7.4.2 中断发生时的压栈 ······308
7.4.3 中断错误码 ······310
7.5 可编程中断控制器 8259A ······311
7.5.1 8259A 介绍 ······311
7.5.2 8259A 的编程 ······314
7.6 编写中断处理程序 ······319
7.6.1 从最简单的中断处理程序开始 ······319
7.6.2 改进中断处理程序 ······335
7.6.3 调试实战：处理器进入中断时压栈出栈完整过程 ······339
7.7 可编程计数器/定时器 8253 简介 ······346
7.7.1 时钟——给设备打拍子 ······346
7.7.2 8253 入门 ······348
7.7.3 8253 控制字 ······349
7.7.4 8253 工作方式 ······350
7.7.5 8253 初始化步骤 ······353
7.8 提高时钟中断的频率，让中断来得更猛烈一些 ······354

第 8 章 内存管理系统 ······357

8.1 makefile 简介 ······357
8.1.1 makefile 是什么 ······357
8.1.2 makefile 基本语法 ······358
8.1.3 跳到目标处执行 ······360
8.1.4 伪目标 ······361
8.1.5 make：递归式推导目标 ······362
8.1.6 自定义变量与系统变量 ······363
8.1.7 隐含规则 ······365
8.1.8 自动化变量 ······366
8.1.9 模式规则 ······367
8.2 实现 assert 断言 ······367
8.2.1 实现开、关中断的函数 ······367
8.2.2 实现 ASSERT ······370
8.2.3 通过 makefile 来编译 ······372
8.3 实现字符串操作函数 ······374
8.4 位图 bitmap 及其函数的实现 ······377
8.4.1 位图简介 ······377
8.4.2 位图的定义与实现 ······378
8.5 内存管理系统 ······381
8.5.1 内存池规划 ······381
8.5.2 内存管理系统第一步，分配页内存 ······388

第 9 章 线程 ······398

9.1 实现内核线程 ······398
9.1.1 执行流 ······398
9.1.2 线程到底是什么 ······399
9.1.3 进程与线程的关系、区别简述 ······402
9.1.4 进程、线程的状态 ······405
9.1.5 进程的身份证——PCB ······405
9.1.6 实现线程的两种方式——内核或用户进程 ······406
9.2 在内核空间实现线程 ······409
9.2.1 简单的 PCB 及线程栈的实现 ······409
9.2.2 线程的实现 ······413
9.3 核心数据结构，双向链表 ······417
9.4 多线程调度 ······421
9.4.1 简单优先级调度的基础 ······421
9.4.2 任务调度器和任务切换 ······425

第 10 章 输入输出系统 ······439

10.1 同步机制——锁 ······439
10.1.1 排查 GP 异常，理解原子操作 ······439
10.1.2 找出代码中的临界区、互斥、竞争条件 ······444
10.1.3 信号量 ······445

10.1.4 线程的阻塞与唤醒 ······447
10.1.5 锁的实现 ······449
10.2 用锁实现终端输出 ······452
10.3 从键盘获取输入 ······456
10.3.1 键盘输入原理简介 ······456
10.3.2 键盘扫描码 ······457
10.3.3 8042 简介 ······463
10.3.4 测试键盘中断处理程序 ······465
10.4 编写键盘驱动 ······468
10.4.1 转义字符介绍 ······468
10.4.2 处理扫描码 ······469
10.5 环形输入缓冲区 ······476
10.5.1 生产者与消费者问题简述 ······476
10.5.2 环形缓冲区的实现 ······478
10.5.3 添加键盘输入缓冲区 ······481
10.5.4 生产者与消费者实例测试 ······482

第 11 章 用户进程 ······485

11.1 为什么要有任务状态段 TSS ······485
11.1.1 多任务的起源，很久很久以前…… ······485
11.1.2 LDT 简介 ······486
11.1.3 TSS 的作用 ······488
11.1.4 CPU 原生支持的任务切换方式 ······492
11.1.5 现代操作系统采用的任务切换方式 ······495
11.2 定义并初始化 TSS ······497
11.3 实现用户进程 ······501
11.3.1 实现用户进程的原理 ······501
11.3.2 用户进程的虚拟地址空间 ······501
11.3.3 为进程创建页表和 3 特权级栈 ······502
11.3.4 进入特权级 3 ······505
11.3.5 用户进程创建的流程 ······506
11.3.6 实现用户进程——上 ······507
11.3.7 bss 简介 ······513
11.3.8 实现用户进程——下 ······515
11.3.9 让进程跑起来——用户进程的调度 ······519
11.3.10 测试用户进程 ······520

第 12 章 进一步完善内核 ······523

12.1 Linux 系统调用浅析 ······523
12.2 系统调用的实现 ······527
12.2.1 系统调用实现框架 ······527
12.2.2 增加 0x80 号中断描述符 ······527
12.2.3 实现系统调用接口 ······528
12.2.4 增加 0x80 号中断处理例程 ······528
12.2.5 初始化系统调用和实现 sys_getpid ······530
12.2.6 添加系统调用 getpid ······531
12.2.7 在用户进程中的系统调用 ······532
12.2.8 系统调用之栈传递参数 ······534
12.3 让用户进程“说话” ······536
12.3.1 可变参数的原理 ······536
12.3.2 实现系统调用 write ······538
12.3.3 实现 printf ······539
12.3.4 完善 printf ······542
12.4 完善堆内存管理 ······545
12.4.1 malloc 底层原理 ······545
12.4.2 底层初始化 ······548
12.4.3 实现 sys_malloc ······550
12.4.4 内存的释放 ······555
12.4.5 实现 sys_free ······558
12.4.6 实现系统调用 malloc 和 free ······562

第 13 章 编写硬盘驱动程序 ······566

13.1 硬盘及分区表 ······566
13.1.1 创建从盘及获取安装的磁盘数 ······566
13.1.2 创建磁盘分区表 ······567
13.1.3 磁盘分区表浅析 ······571
13.2 编写硬盘驱动程序 ······578
13.2.1 硬盘初始化 ······578
13.2.2 实现 thread_yield 和 idle 线程 ······582
13.2.3 实现简单的休眠函数 ······584
13.2.4 完善硬盘驱动程序 ······585
13.2.5 获取硬盘信息，扫描分区表 ······590

第 14 章 文件系统 ······595

14.1 文件系统概念简介 ······595
14.1.1 inode、间接块索引表、文件控制块 FCB 简介 ······595
14.1.2 目录项与目录简介 ······597
14.1.3 超级块与文件系统布局 ······599
14.2 创建文件系统 ······601
14.2.1 创建超级块、i 结点、目录项 ······601
14.2.2 创建文件系统 ······603
14.2.3 挂载分区 ······609

14.3 文件描述符简介 ······612
14.3.1 文件描述符原理 ······612
14.3.2 文件描述符的实现 ······614
14.4 文件操作相关的基础函数 ······615
14.4.1 inode 操作有关的函数 ······616
14.4.2 文件相关的函数 ······620
14.4.3 目录相关的函数 ······623
14.4.4 路径解析相关的函数 ······628
14.4.5 实现文件检索功能 ······630
14.5 创建文件 ······633
14.5.1 实现 file_create ······633
14.5.2 实现 sys_open ······636
14.5.3 在文件系统上创建第 1 个文件 ······639
14.6 文件的打开与关闭 ······640
14.6.1 文件的打开 ······640
14.6.2 文件的关闭 ······642
14.7 实现文件写入 ······643
14.7.1 实现 file_write ······643
14.7.2 改进 sys_write 及 write 系统调用 ······648
14.7.3 把数据写入文件 ······650
14.8 读取文件 ······651
14.8.1 实现 file_read ······651
14.8.2 实现 sys_read 与功能验证 ······653
14.9 实现文件读写指针定位功能 ······655
14.10 实现文件删除功能 ······657
14.10.1 回收 inode ······657
14.10.2 删除目录项 ······660
14.10.3 实现 sys_unlink 与功能验证 ······663
14.11 创建目录 ······665
14.11.1 实现 sys_mkdir 创建目录 ······666
14.11.2 创建目录功能验证 ······669
14.12 遍历目录 ······671
14.12.1 打开目录和关闭目录 ······671
14.12.2 读取 1 个目录项 ······673
14.12.3 实现 sys_readdir 及 sys_rewinddir ······674
14.13 删除目录 ······676
14.13.1 删除目录与判断空目录 ······676
14.13.2 实现 sys_rmdir 及功能验证 ······677
14.14 任务的工作目录 ······679
14.14.1 显示当前工作目录的原理及基础代码 ······679
14.14.2 实现 sys_getcwd ······681
14.14.3 实现 sys_chdir 改变工作目录 ······683
14.15 获得文件属性 ······684
14.15.1 ls 命令的幕后功臣 ······684
14.15.2 实现 sys_stat ······685

第 15 章 系统交互 ······687

15.1 fork 的原理与实现 ······687
15.1.1 什么是 fork ······687
15.1.2 fork 的实现 ······689
15.1.3 添加 fork 系统调用与实现 init 进程 ······695
15.2 添加 read 系统调用，获取键盘输入 ······696
15.3 添加 putchar、clear 系统调用 ······697
15.4 实现一个简单的 shell ······699
15.4.1 shell 雏形 ······699
15.4.2 添加 Ctrl+u 和 Ctrl+l 快捷键 ······701
15.4.3 解析键入的字符 ······703
15.4.4 添加系统调用 ······705
15.4.5 路径解析转换 ······708
15.4.6 实现 ls、cd、mkdir、ps、rm 等命令 ······712
15.5 加载用户进程 ······717
15.5.1 实现 exec ······717
15.5.2 让 shell 支持外部命令 ······723
15.5.3 加载硬盘上的用户程序执行 ······724
15.5.4 使用户进程支持参数 ······727
15.6 实现系统调用 wait 和 exit ······731
15.6.1 wait 和 exit 的作用 ······731
15.6.2 孤儿进程和僵尸进程 ······732
15.6.3 一些基础代码 ······733
15.6.4 实现 wait 和 exit ······737
15.6.5 实现 cat 命令 ······741
15.7 管道 ······745
15.7.1 管道的原理 ······745
15.7.2 管道的设计 ······747
15.7.3 管道的实现 ······748
15.7.4 利用管道实现进程间通信 ······752
15.7.5 在 shell 中支持管道 ······754

参考文献 ······760

第 0 章　一些你可能正感到迷惑的问题

正如计算机中数组下标是从 0 开始的，我们的内容也从 0 开始，尽量做到低基础学习（负责地说，不是 0 基础，而且还只是尽量），解释一些学习过程中经常被问到的问题。

0.1 操作系统是什么

我并没有给你提供教科书上对操作系统的定义，因为解释得太抽象了，看了之后似乎只是获得一些感性认识，好奇心强的读者反而会产生更多迷惑。为了说清楚问题，让我给您举个例子。

让我们扯点远的……在盘古开天之际，除动物以外，世界上只有土地、荒草、树木、石头等资源。人们为了躲避天灾、野兽攻击等危险，开始住进了山洞，为了获取食物，用石头和树木等材料打造一些武器。当时所有人都在做这些相同的事。这就是没有组织的人类社会，所有人都在重复“造轮子”。

后来各个地区有了自己权威性的部落，部落都专门找人打造武器，谁需要武器就直接申请领取便可，大部分人不需要自己打造武器了。后来嫌打猎太麻烦了，干脆养一些家畜好了，直接供给人们，谁需要可以过来交换。这就是把大家的重复性劳动集中到了一起，让人们可以专注于自己的事情。

再后来，部落之间为了通信，开始有信使了，这是最原始的通信方式。到后来发展到有社会组织，通信越来越频繁了，干脆搞个驿站吧，谁需要通信，直接写信，由驿站代为送达。

随着人口越来越多，社会组织需要了解到底有多少人，为了方便人口管理，于是就在各地建了“户籍办事”处，人们的生老病死都要到那里登记申报。

说到这我估计您已经猜出我所说的了，上面提到的部落其实就是最原始的操作系统雏形，它将大家都需要的工作找专人负责，大家不用重复劳动。而以上的社会组织其实就是代表现代操作系统，除了把重复性工作集中化，还有了自己的管理策略。

把上面的例子再具体一下，人们想狩猎时，可不可以自己先打造武器，然后拿着自己的武器去狩猎？当然可以，自己制造武器完全没有问题，但部落既然有现成的武器可用，何必自己再费事呢。另外，部落担不担心你随意制造武器会对他人造成伤害？当然会，所以部落不允许你自己制造武器了，人们只有申请的资格，给不给还是要看人家部落的意愿。这就是操作系统提供给用户进程一些系统调用，当用户进程需要某个资源时，直接调用便可，不用自己再费尽心思考虑硬件的事情了，由操作系统把资源获取到后交给用户进程，用户进程可以专注于自己的工作。但操作系统为了保护计算机系统不被损害，不允许用户进程直接访问硬件资源，比如用户进程将操作系统所占据的内存恶意覆盖了，操作系统也就不复存在了，没有操作系统的话，计算机将会瘫痪无法运作。

当人们想和远方的朋友说话时，虽然可以徒步走到亲朋好友身边再对其表达想说的话，但社会组织已经给提供了邮局和电话，何必自己再大老远跑一趟呢。这就是操作系统（社会组织）提供的资源。两个人想在一起生活，要不要一定先结婚呢？完全不用，领不领证都不会阻碍人们在一起生活，但是社会组织为了方便人口管理做了额外约束。不领证的话，至少社会组织无法预测未来人口数量趋势，无法做出宏观调控，甚至这是找到你家人的一种方法。这就如 Linux 系统中的内存管理，分别要记录哪些页是 Active，哪些是“脏页”。不记录会不会影响程序执行，当然不会，记录这些状态还不是为了更好地管理内存吗。

以上说的社会组织和人们之间的关系，正是操作系统和用户进程的关系，希望大家能对操作系统有个初步印象，后面的实践中我们将实例化各个部分。

0.2 你想研究到什么程度

学无止境，学习没有说到头的那天。学习到任何程度都是存有疑惑的，就像中学和大学都讲物理，但学的深度不一样，各个阶段都会产生疑问。我们只是基于一些公认的知识，使其作为学习的起点，并以此展开上层的研究。

比如我对太空很感兴趣，大伙儿都知道地球围绕太阳做周期性公转，后来又知道电子围绕原子核来做周期性公转运动，这和地球绕太阳公转的行为如出一辙，甚至我在想太阳是不是相当于原子核，地球相当于一个电子，我们只是生活在一个电子上……而我们身体里有那么多的原子和电子，对那些我们身体中更为细微的生物来说，我们的身体是不是一个宇宙，无尽的猜想，无尽的疑惑。想法虽然有些荒诞，但基于现有科技目前谁也无法证明这是错的，而且近期已有科学文献证明人的大脑就像个宇宙。如果无止境地刨根问底下去，虽然会对底层科学更加清晰，但这对上层知识的学习非常不利，从而我们需要一个公设，我们认为原子是不可再分的，没有更微小的对象了，一切理论研究以此为基础展开。比如乘法是基于加法的，我们研究 3×4 等于多少，必须要承认 1+1 等于 2，并认为其为真理，不用再去质疑 1+1 为什么等于 2 了，这就是我们的公设，至于为什么 1+1 等于 2，还是由专门研究基础科学的学者们去探究吧。

学习操作系统也一样，不必纠结于硬件内部是如何工作的，我们只要认为给硬件一个输入，硬件就会给我一个输出就行了，因为即使你学到了硬件内部电子电路，随着你不断进步，钻研不断深入，也许有一天你的求知欲到了物理领域，并产生了物理科学方面的质疑……这让我想到一个笑话，某人准备去买自行车，结果被销售人员不断劝说，加点钱就能买摩托啦，等决定买摩托时，销售人员又说既然都决定买摩托车了，不如再加点买汽车吧，给出了各种汽车方面的优势，欲望需求不断升级，不断被销售劝说，最后居然花了几百万元买车，最后才想起自己是来买自行车的，甚至他还没有驾照……于是，咱们赶紧就此打住，我们是来学操作系统的。

你想学到哪个程度呢，你的公设是什么，要不咱们还是走一步说一步吧。

0.3 写操作系统，哪些需要我来做

首先应该明确，在计算机中有分层的概念，也就是说，计算机是一个大的组合物，由各个部分组合成一个系统。每个部分就是一层功能模块，各司其职，它只完成一定的工作，并将自己的工作结果（也就是输出）交给下一层的模块，这里的模块指的是各种外设、硬件。

这样，各种工作成果不断累加，通过这种流水线式的上下游协作，便实现了所谓的系统。可见，系统就是各种功能组合到一起后，产生最终输出的组合物。就像人的身体，胃负责搅拌食物，将这些食物变食糜后交给小肠，因为小肠只能处理流食，所以上游的输出一定要适合作为下游的输入，是不是有点类似管道操作了，哈哈，分工协作是大自然的安排，并不是只有计算机世界才有。我们人类的思想是大自然安排好的，所以人类创造的事物也是符合大自然规律的。

好，赶紧回到正题，操作系统是管理资源的软件，操作系统能做什么，取决于主机上硬件的功能。就像用 Maya 造一个人体模型出来，首先我得知道 Maya 这个软件提供曲线曲面各种建模方法才行，换句话说，对于人体建模，你不可能会想到用 QQ，因为它不是干这个的。我想说的是硬件不支持的话，操作系统也没招……操作系统一直是所谓的底层，拥有至高无上的控制权，一副牛气轰轰的样子，原来也要依仗他人啊。是啊，操作系统毕竟是软件，而软件的逻辑是需要作用在硬件上才能体现出来的。

所以说，写操作系统需要了解硬件，这些硬件提供了软件方面的接口，这样我们的操作系统通过软件（计算机指令）就能够控制硬件。我们需要做的就是知道如何通过计算机指令来控制硬件，参考硬件手册这下少不了啦。

0.4 软件是如何访问硬件的

硬件是各种各样的，发展速度还是非常快的。各个硬件都有自己的个性，操作系统不可能及时更新各

种硬件的驱动方法吧。比如，刚出来某个新硬件，OS 开发者们便开始为其写驱动，这不太现实，会把人累死的。于是乎，便出现了各种硬件适配设备，这就是 IO 接口。接口其实就是标准，大家生产出来的硬件按照这个标准工作就实现了通用。

硬件在输入输出上大体分为串行和并行，相应的接口也就是串行接口和并行接口。串行硬件通过串行接口与 CPU 通信，反过来也是，CPU 通过串行接口与串行设备数据传输。并行设备的访问类似，只不过是通过并行接口进行的。

访问外部硬件有两个方式。

（1）将某个外设的内存映射到一定范围的地址空间中，CPU 通过地址总线访问该内存区域时会落到外设的内存中，这种映射让 CPU 访问外设的内存就如同访问主板上的物理内存一样。有的设备是这样做的，比如显卡，显卡是显示器的适配器，CPU 不直接和显示器交互，它只和显卡通信。显卡上有片内存叫显存，它被映射到主机物理内存上的低端 1MB 的 0xB8000～0xBFFFF。CPU 访问这片内存就是访问显存，往这片内存上写字节便是往屏幕上打印内容。看上去这么高大上的做法是怎么实现的，这个我们就不关心了，前面说过，计算机中处处是分层，我们要充分相信上一层的工作。

（2）外设是通过 IO 接口与 CPU 通信的，CPU 访问外设，就是访问 IO 接口，由 IO 接口将信息传递给另一端的外设，也就是说，CPU 从来不知道有这些设备的存在，它只知道自己操作的 IO 接口，你看，处处体现着分层。

于是问题来了，如何访问到 IO 接口呢，答案就是 IO 接口上面有一些寄存器，访问 IO 接口本质上就是访问这些寄存器，这些寄存器就是人们常说的端口。这些端口是人家 IO 接口给咱们提供的接口。人家接口电路也有自己的思维（系统），看到寄存器中写了什么就做出相应的反应。接口提供接口，哈哈，有意思。不过这是人家的约定，没有约定就乱了，各干各的，大家都累，咱们只要遵循人家的规定就能访问成功。

0.5 应用程序是什么，和操作系统是如何配合到一起的

应用程序是软件（似乎是废话，别急，往后看），操作系统也是软件。CPU 会将它们一视同仁，甚至，CPU 不知道自己在执行的程序是操作系统，还是一般应用软件，CPU 只知道去 cs：ip 寄存器中指向的内存取指令并执行，它不知道什么是操作系统，也无需知道。

操作系统是人想出来的，为了让自己管理计算机方便而创造出来的一套管理办法。

应用程序要用某种语言编写，而语言又是编译器来提供的。其实根本就没有什么语言，有的只是编译器。是编译器决定怎样解释某种关键字及某种语法。语言只是编译器和大家的约定，只要写入这样的代码，编译器便将其翻译成某种机器指令，翻译成什么样取决于编译器的行为，和语言无关，比如说 C 语言的 printf 函数，它的功能不是说一定要把字符打印到屏幕上，这要看编译器对这种关键字的处理。

编译器提供了一套库函数，库函数中又有封装的系统调用，这样的代码集合称之为运行库。C 语言的运行库称为 C 运行库，就是所谓的 CRT（C Runtime Library）。

应用程序加上操作系统提供功能才算是完整的程序。由于有了操作系统的支持，一些现成的东西已经摆在那了，但这些是属于操作系统的，不是应用程序的，所以咱们平时所写的应用程序只是半成品，需要调用操作系统提供好的函数才能完整地做成一件事，而这个函数便是系统调用。

用户态与内核态是对 CPU 来讲的，是指 CPU 运行在用户态（特权 3 级）还是内核态（特权 0 级），很多人误以为是对用户进程来讲的。

用户进程陷入内核态是指：由于内部或外部中断发生，当前进程被暂时终止执行，其上下文被内核的中断程序保存起来后，开始执行一段内核的代码。是内核的代码，不是用户程序在内核的代码，用户代码怎么可能在内核中存在，所以“用户态与内核态”是对 CPU 来说的。

当应用程序陷入内核后，它自己已经下 CPU 了，以后发生的事，应用程序完全不知道，它的上下文环境已经被保存到自己的 0 特权级栈中了，那时在 CPU 上运行的程序已经是内核程序了。所以要清楚，内核代码并不是成了应用程序的内核化身，操作系统是独立的部分，用户进程永远不会因为进入内核态而变身为操作系统了。

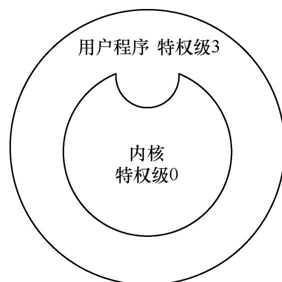
应用程序是通过系统调用来和操作系统配合完成某项功能的，有人可能会问：我写应用程序时从来没写什么系统调用的代码啊。这是因为你用到的标准库帮你完成了这些事，库中提供的函数其实都已经封装好了系统调用，你需要跟下代码才会看到。其实也可以跨过标准库直接执行系统调用，对于 Linux 系统来说，直接嵌入汇编代码“int 0x80”便可以直接执行系统调用，当然要提前设置好系统调用子功能号，该子功能号用寄存器 eax 存储。

会不会有人又问，编译器怎么知道系统调用接口是什么，哈哈，您想啊，下载编译器时，是不是要选择系统版本，编译器在设计时也要知道自己将来运行在哪个系统平台上，所以这都是和系统绑定好的，各个操作系统都有自己的系统调用号，编译器厂商在代码中已经把宿主系统的系统调用号写死了，没什么神奇的。

0.6 为什么称为“陷入”内核

前面提到了用户进程陷入内核，这个好解释，如果把软件分层的话，最外圈是应用程序，里面是操作系统，如图 0-1 所示。

应用程序处于特权级 3，操作系统内核处于特权级 0。当用户程序欲访问系统资源时（无论是硬件，还是内核数据结构），它需要进行系统调用。这样 CPU 便进入了内核态，也称管态。看图中凹下去的部分，是不是有陷进去的感觉，这就是“陷入内核”。



▲图 0-1　陷入内核

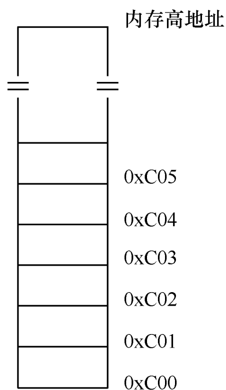
0.7 内存访问为什么要分段

按理说咱们应该先看看段是什么，不过了解段是什么之前，先看看内存是什么样子，如图 0-2 所示。

内存按访问方式来看，其结构就如同上面的长方形带子，地址依次升高。为了解释问题更明白，我们假设还在实模式下，如果读者不清楚什么是实模式也不要紧，这并不影响理解段是什么，故暂且先忽略。

内存是随机读写设备，即访问其内部任何一处，不需要从头开始找，只要直接给出其地址便可。如访问内存 0xC00，只要将此地址写入地址总线便可。问题来了，分段是内存访问机制，是给 CPU 用的访问内存的方式，只有 CPU 才关注段，那为什么 CPU 要用段呢，也就是为什么 CPU 非得将内存分成一段一段的才能访问呢？

说来话长，现实行业中有很多问题都是历史遗留问题，计算机行业也不能例外。分段是从 CPU 8086 开始的，限于技术和经济，那时候电脑还是非常昂贵的东西，所以 CPU 和寄存器等宽度都是 16 位的，并不是像今天这样寄存器已经扩展到 64 位，当然编译器用的最多的还是 32 位。16 位寄存器意味着其可存储的数字范围是 2 的 16 次方，即 65536 字节，64KB。那时的计算机没有虚拟地址之说，只有物理地址，访问任何存储单元都直接给出物理地址。



▲图 0-2　内存示例

编译器在编译程序时，肯定要根据 CPU 访问内存的规则将代码编译成机器指令，这样编译出来的程序才能在该 CPU 上运行无误，所以说，在直接以绝对物理地址访问内存的 CPU 上运行程序，该程序中指令的地址也必须得是绝对物理地址。总之，要想在该硬件上运行，就要遵从该硬件的规则，操作系统和编译器也无一例外。

若加载程序运行，不管其是内核程序，还是用户程序，程序中的地址若都是绝对物理地址，那该程序必须放在内存中固定的地方，于是，两个编译出来地址相同的用户程序还真没法同时运行，只能运行一个。于是伟大的计算机前辈们用分段的方式解决了这一问题，让 CPU 采用“段基址+段内偏移地址”的方式来访问任意内存。这样的好处是程序可以重定位了，尽管程序指令中给的是绝对物理地址，但终究可以同时运行多个程序了。

什么是重定位呢，简单来说就是将程序中指令的地址改写成另外一个地址，但该地址处的内容还是原

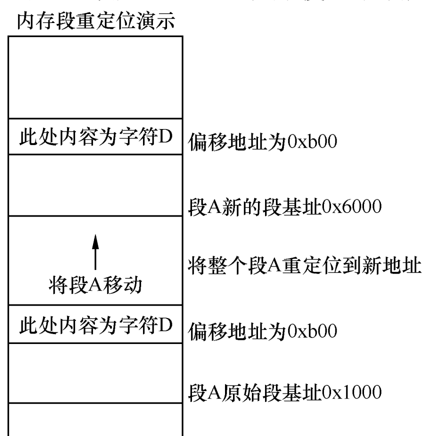
地址处的内容。

CPU 采用“段基址+段内偏移地址”的形式访问内存，就需要专门提供段基址寄存器，这些是 cs、ds、es 等。程序中需要用到哪块内存，只要先加载合适的段到段基址寄存器中，再给出相对于该段基址的偏移地址便可，CPU 中的地址单元会将这两个地址相加后的结果用于内存访问，送上地址总线。

注意，很多读者都觉得段基址一定得是 65536 的倍数（16 位段基址寄存器的容量），这个真的不用，段基址可以是任意的。这就是段可以重叠的原因。

举个例子，看图 0-2，假设段基址为 0xC00，要想访问物理内存 0xC01，就要将用 0xC00：0x01 的方式来访问才行。若将段基址改为 0xc01，还是访问 0xC01，就要用 0xC01：0x00 的方式来访问。同样，若想访问物理内存 0xC04，段基址和段内偏移的组合可以是：0xC01：0x03、0xC02：0x02、0xC00：0xC04 等，总之要想访问某个物理地址，只要凑出合适的段基地址和段内偏移地址，其和为该物理地址就行了。这时估计有人会问这样行不行，0xC05：-1，能这样提问的同学都是求知欲极强的，可以自己试一下。

说了这么多，我想告诉你的是只要程序分了段，把整个段平移到任何位置后，段内的地址相对于段基址是不变的，无论段基址是多少，只要给出段内偏移地址，CPU 就能访问到正确的指令。于是加载用户程序时，只要将整个段的内容复制到新的位置，再将段基址寄存器中的地址改成该地址，程序便可准确无误地运行，因为程序中用的是段内偏移地址，相对于新的段基址，该偏移地址处的内存内容还是一样的，如图 0-3 所示。



▲图 0-3　段的重定位

所以说，程序分段首先是为了重定位，我说的是首先，下面还有其他理由呢。

偏移地址也要存入寄存器，而那时的寄存器是 16 位的，也就是一个段最多可以访问到 64KB。而那时的内存再小也有 1MB，改变段基址，由一个段变为另一个段，就像一个段在内存中飘移，采用这种在内存中来回挪位置的方式可以访问到任意内存位置。

所以说，程序分段又是为了将大内存分成可以访问的小段，通过这样变通的方法便能够访问到所有内存了。

但想一想，1M 是 2 的 20 次方，1MB 内存需要 20 位的地址才能访问到，如何做到用 16 位寄存器访问 20 位地址空间呢？

在 8086 的寻址方式中，有基址寻址，这是用基址寄存器 bx 或 bp 来提供偏移地址的，如“mov [bx]，0x5;”指令便是将立即数 0x5 存入 ds：bx 指向的内存。

大家看，bx 寄存器是 16 位的，它最大只能表示 0～0xFFFF 的地址空间，即 64KB，也就是单一的一个寄存器无法表示 20 位的地址空间——1MB。也许有人会说，段基址和段内偏移地址都搞到最大，都为 0xFFFF，对不起，即使不溢出的话，其结果也只是由 16 位变成了 17 位，即两个 n 位的数字无论多大，其相加的结果也超不过 n+1 位，因为即使是两个相同的数相加，其结果相当于乘以 2，也就是左移一位而已，依然无法访问 20 位的地址空间。也许读者又有好建议了：CPU 的寻址方式又不是仅仅这一种，上面的限制是因为寄存器是 16 位，只要不全部通过寄存器不就行了吗。既然段寄存器必须得用，那就在偏移地址上下功夫，不要把偏移地址写在寄存器里了，把它直接写成 20 位立即数不就行啦。例如 mov ax，[0x12345]，这样最终的地址是 ds+0x12345，肯定是 20 位，解决啦。不错，这种是直接寻址方式，至少道理上讲得通，这是通过编程技巧来突破这一瓶颈的，能想到这一点我觉得非常 nice。但是作为一个严谨的 CPU，既然宣称支持了通过寄存器来寻址，那就要能够自圆其说才行，不能靠程序员的软实力来克服 CPU 自身的缺陷。于是，一个大胆的想法出现了。

16 位的寄存器最多访问到 64KB 大小的内存。虽然 1MB 内存中可容纳 1MB/64KB=16 个最大段，但这只是可以容纳而已，并不是说可以访问到。16 位的寄存器超过 0xffff 后将会回卷到 0，又从 0 重新开始。20 位宽度的内存地址空间必然只能由 20 位宽度的地址来访问。问题又来了，在当时只有 16 位寄存器的情况下是如何做到访问 20 位地址空间的呢？

这是因为 CPU 设计者在地址处理单元中动了手脚，该地址部件接到“段基址+段内偏移地址”的地址后，自动将段基址乘以 16，即左移了 4 位，然后再和 16 位的段内偏移地址相加，这下地址变成了 20 位了吧，行

啦，有了 20 位的地址便可以访问 20 位的空间，可以在 1MB 空间内自由翱翔了。

0.8 代码中为什么分为代码段、数据段？这和内存访问机制中的段是一回事吗

首先，程序不是一定要分段才能运行的，分段只是为了使程序更加优美。就像用饭盒装饭菜一样，完全可以将很多菜和米饭混合在一起，或者搅拌成一体，哈哈，但这样可能就没什么胃口啦。如果饭盒中有好多小格子，方便将不同的菜和饭区分存放，这样会让我们胃口大开增加食欲。

x86 平台的处理器是必须要用分段机制访问内存的，正因为如此，处理器才提供了段寄存器，用来指定待访问的内存段起始地址。我们这里讨论的程序代码中的段（用 section 或 segment 来定义的段，不同汇编编译器提供的关键字有所区别，功能是一样的）和内存访问机制中的段本质上是一回事。在硬件的内存访问机制中，处理器要用硬件——段寄存器，指向软件——程序代码中用 section 或 segment 以软件形式所定义的内存段。

分段是必然的，只是在平坦模型下，硬件段寄存器中指向的内存段为最大的 4GB，而在多段模式下编程，硬件段寄存器中指向的内存段大小不一。

对于在代码中的分段，有的是操作系统做的，有的是程序员自己划分的。如果是在多段模型下编程，我们必然会在源码中定义多个段，然后需要不断地切换段寄存器所指向的段，这样才能访问到不同段中的数据，所以说，在多段模型下的程序分段是程序员人为划分的。如果是在平坦模型下编程，操作系统将整个 4GB 内存都放在同一个段中，我们就不需要来回切换段寄存器所指向的段。对于代码中是否要分段，这取决于操作系统是否在平坦模型下。

一般的高级语言不允许程序员自己将代码分成各种各样的段，这是因为其所用的编译器是针对某个操作系统编写的，该操作系统采用的是平坦模型，所以该编译器要编译出适合此操作系统加载运行的程序。由于处理器支持了具有分页机制的虚拟内存，操作系统也采用了分页模型，因此编译器会将程序按内容划分成代码段和数据段，如编译器 gcc 会把 C 语言写出的程序划分成代码段、数据段、栈段、.bss 段、堆等部分。这会由操作系统将编译器编译出来的用户程序中的各个段分配到不同的物理内存上。对于目前咱们用高级语言编码来说，我们之所以不用关心如何将程序分段，正是由于编译器按平坦模型编译，而程序所依赖的操作系统又采用了虚拟内存管理，即处理器的分页机制。像汇编这种低级语言允许程序员为自己的程序分段，能够灵活地编排布局，这就属于人为将程序分成段了，也就是采用多段模型编程。

这么说似乎不是很清楚，一会再用例子和大伙儿解释就明白了。在这之前，先和大家明确一件事。

CPU 是个自动化程度极高的芯片，就像心脏一样，给它一个初始的收缩，它将永远地跳下去。突然想到 Intel 的广告词：给你一颗奔腾的心。

只要给出 CPU 第一个指令的起始地址，CPU 在它执行本指令的同时，它会自动获取下一条的地址，然后重复上述过程，继续执行，继续取址。假如执行的每条指令都正确，没有异常发生的话，我想它可以运行到世界的尽头，能让它停下来的唯一条件就是断电。

它为什么能够取得下一条指令地址？也就是说为什么知道下一条指令在哪里。这是因为程序中的指令都是挨着的，彼此之间无空隙。有同学可能会问，程序中不是有对齐这回事吗？为了对齐，编译器在程序中塞了好多 0。是的，对齐确实是让程序中出现了好多空隙，但这些空隙是数据间的空隙，指令间不存在空隙，下一条指令的地址是按照前面指令的尺寸大小排下来的，这就是 Intel 处理器的程序计数器 cs：eip 能够自动获得下一条指令的原理，即将当前 eip 中的地址加上当前指令机器码的大小便是内存中下一条指令的起始地址。即使指令间有空隙或其他非指令的数据，这也仅仅是在物理上将其断开了，依然可以用 jmp 指令将非指令部分跳过以保持指令在逻辑上连续，我们在后面会通过实例验证这一原理。

为了让程序内指令接连不断地执行，要把指令全部排在一起，形成一片连续的指令区域，这就是代码段。这样 CPU 肯定能接连不断地执行下去。指令是由操作码和操作数组成的，这对于数据也一样，程序运行不仅要有操作码，也得有操作数，操作数就是指程序中的数据。把数据连续地并排在一起存储形成的

段落，就称为数据段。

指令大小是由实际指令的操作码决定的，也就是说 CPU 在译码阶段拿到了操作码后，就知道实际指令所占的大小。其实说来说去，本质上就是在解释地址是怎么来的。这部分在第 3 章中的“什么是地址”节中有详解。

给大家演示个小例子，代码没有实际意义，是我随便写的，只是为方便大家理解指令的地址，代码如下。

code_seg.S

```
1     mov ds,ax
2     mov ax,[var]
3 label:
4     jmp label
5     var dw 0x99
```

本示例一共就 5 行，简单纯粹为演示。将其编译为二进制文件，程序内容是：

```
8E D8 A1 07 00 EB FE 99 00
```

就这 9 个字节的内容，有没有觉得一阵晕炫。如果没有，目测读者兄弟的技术水平远在我之上，请略过本书。

其实这 9 个字节的内容就是机器码。为了让大家理解得更清晰，给大家列个机器码和源码对照表，见表 0-1。

表 0-1　　机器码和源码对照表

地　　址	机　器　码	源　　码
00000000	8ED8	mov ds，ax
00000002	A10700	mov ax，[0x7]
00000005	EBFE	jmp short 0x5
00000007		var dw 0x99
00000008		

表 0-1 第 1 行，地址 0 处的指令是“mov ds，ax”，其机器码是 8ED8，这是十六进制表示，可见其大小是 2 字节。前面说过，下一条指令的地址是按照前面指令的尺寸排下来的，那第 2 行指令的起始地址是 0+2=2。在第 2 行的地址列中，地址确实是 2。这不是我故意写上去的，编译器真的就是这样编排的。第 2 列的指令是“mov ax，[0x7]”（0x7 是变量 var 经过编译后的地址），其机器码是 A10700，这是 3 字节大小。所以第 3 条指令的地址是 2+3=5。后面的指令地址也是这样推算的。程序虽然很短，但麻雀虽小，五脏俱全，完美展示了程序中代码紧凑无隙的布局。

现在大伙儿明白为什么 CPU 能源源不断获取到指令了吧，如前所述，原因首先是指令是连续紧凑的，其次是通过指令机器码能够判断当前指令长度，当前指令地址+当前指令长度=下一条指令地址。

上面给出的例子，其指令在物理上是连续的，其实在 CPU 眼里，只要指令逻辑上是连续的就可以，没必要一定得是物理上连续。所以，明确一点，即使数据和代码在物理上混在一起，程序也是可以运行的，这并不意味着指令被数据“断开”了。只要程序中有指令能够跨过这些数据就行啦，最典型的就是用 jmp 跳过数据区。

比如这样的汇编代码：

```
1     jmp start         ;跳转到第三行的 start，这是 CPU 直接执行的指令
2     var dd  1         ;定义变量 var 并赋值为 1。分配变量不是 CPU 的工作
                        ;汇编器负责分配空间并为变量编址
3     start:            ;标号名为 start，会被汇编器翻译为某个地址
4     mov ax,0       ； 将 ax 赋值为 0
```

这几行代码没有实际意义，只是为了解释清楚问题，咱们只要关注在第 2 行的定义变量 var 之前为什么要 jmp start。如果将上面的汇编代码按纯二进制编译，如果不加第 1 行的 jmp，CPU 也许会发出异常，显示无效指令，也许不知道执行到哪里去了。因为 CPU 只会执行 cs：ip 中的指令，这两个寄存器记录的是下一条待执行指令的地址，下一个地址 var 处的值为 1，显然我们从定义中看出这只是数据，但指令和数据都是二进制数字，CPU 可分不出这是指令，还是数据。保不准某些“数据”误打误撞恰恰是某种指令也说不定。既然 var 是我们定义的数据，那么必须加上 jmp start 跳过这个 var 所占的空间才可以。

加个 jmp 指令，这样做一点都不影响运行，只不过这样写出来的程序，其中引用的地址大部分是不连续的，也就是程序在取地址时会显得跳来跳去。就美观层面上看，这样的结构显得很凌乱，不利于程序员阅读与维护。如果把第 2 行的 var 换到第 1 行，数据和代码就分开了，没有混在一起，标号都不用了，代码简洁多了，如下。

```
var dd 1
mov ax,0
```

做过开发的同学都清楚，尽量把同一属性的数据放在一起，这样易于维护。这一点类似于 MVC，在程序逻辑中把模型、视图、控制这三部分分开，这样更新各部分时，不会影响到其他模块。

将数据和代码分开的好处有三点。

第一，可以为它们赋予不同的属性。

例如数据本身是需要修改的，所以数据就需要有可写的属性，不让数据段可写，那程序根本就无法执行啦。程序中的代码是不能被更改的，这样就要求代码段具备只读的属性。真要是在运行过程中程序的下一条指令被修改了，谁知道会产生什么样的灾难。

第二，为了提高 CPU 内部缓存的命中率。

大伙儿知道，缓存起作用的原因是程序的局部性原理。在 CPU 内部也有缓存机制，将程序中的指令和数据分离，这有利于增强程序的局部性。CPU 内部有针对数据和针对指令的两种缓存机制，因此，将数据和代码分开存储将使程序运行得更快。

第三，节省内存。

程序中存在一些只读的部分，比如代码，当一个程序的多个副本同时运行时（比如同时执行多个 ls 命令时），没必要在内存中同时存在多个相同的代码段，这将浪费有限的物理内存资源，只要把这一个代码段共享就可以了。

后两点较容易理解，咱们深入讨论下第一点，不知您有没有想过，数据段或代码段的属性是谁给添加上的呢，是谁又去根据属性保护程序的呢，是程序员吗？是编译器吗？是操作系统吗？还是 CPU 一级的硬件支持？

首先肯定不是程序员，人家操作系统设计人员为了让程序员编写程序更加容易，肯定不会让他们分心去处理这些与业务逻辑无关的事。看看编译器为我们做了什么，它将程序中那些只读的代码编译出来后，放在一片连续的区域，这个区域叫代码段。将那些已经初始化的数据也放在一片连续的区域，这个区域叫数据段，那些具有全局属性的但又未初始化的数据放在 bss 段。总之，程序中段的类型可多了，用“readelf –e elf”命令便可以看到很多段的类型，感兴趣的读者请自行查阅。好了，编译器的工作到此就完事了，显然，数据段和代码段的属性到现在还没有体现出来。

先看 CPU 为我们提供了哪些原生的支持。在保护模式下，有这样一个数据结构，它叫全局描述符表（Global Descriptor Table，GDT），这个表中的每一项称为段描述符。先递归学习一下，什么是描述符？描述符就是描述某种数据的数据结构，是元信息，属于数据的数据。就像人们的身份证，上面有写性别、出生日期、地址等描述个人情况的信息。在段描述符中有段的属性位，在以后的章节中可以看到，其实是有 2 个，一个是 S 字段，占 1bit 大小，另外一个是占 4bit 大小的 TYPE 字段，这两个字段配合在一起使用就能组合出各种属性，如只读、向下扩展、只执行等。提供归提供，可得有人去填写这张表啊，谁来做这事呢，有请操作系统登场。

接着看操作系统为我们做了什么。

操作系统在让 CPU 进入保护模式之前，首先要准备好 GDT，也就是要设置好 GDT 的相关项，填写好段描述符。段描述符填写成什么样，段具备什么样的属性，这完全取决于操作系统了，在这里大家只要知道，段描述符中的 S 字段和 TYPE 字段负责该段的属性，也就是该属性与安全相关。

说到这里，答案似乎浮出水面了。

（1）编译器负责挑选出数据具备的属性，从而根据属性将程序片段分类，比如，划分出了只读属性的代码段和可写属性的数据段。再补充一下，编译器并没有让段具备某种属性，对于代码段，编译器所做的只是将代码归类到一起而已，也就是将程序中的有关代码的多个 section 合并成一个大的 segment（这就是我们所说的代码段），它并没有为代码段添加额外的信息。

（2）操作系统通过设置 GDT 全局描述符表来构建段描述符，在段描述符中指定段的位置、大小及属性（包括 S 字段和 TYPE 字段）。也就是说，操作系统认为代码应该是只读的，所以给用来指向代码段的那个段描述符设置了只读的属性，这才是真正给段添加属性的地方。

（3）CPU 中的段寄存器提前被操作系统赋予相应的选择子（后面章节会讲什么是选择子，暂时将其理解为相当于段基址），从而确定了指向的段。在执行指令时，会根据该段的属性来判断指令的行为，若有返回则发出异常。

总之，编译器、操作系统、CPU 三个配合在一起才能对程序保护，检测出指令中的违规行为。如果 GDT 中的代码段描述符具备可写的属性，那编译器再怎么划分代码段都没有用，有判断权利的只有 CPU。

好，现在大家对 GDT 有个感性认识，随着以后章节中讲 GDT 的时候，大家就会有深刻的理解了。

以上说明了程序按内容分段的原因，那么编译器编译出来的段和内存访问中的段是一回事吗？

其实算一回事，也不算一回事。怎么说呢，我觉得当初 Intel 公司在设计 CPU 时，其采用分段机制访问内存的原因，肯定不是为了上层软件的优美，毕竟那只是逻辑上的东西。那为什么也算一回事呢？

分析一下，编译出来的代码段是指一片连续的内存区域。这个段有自己的起始地址，也有自己的大小范围。用户进程中的段，只是为了便于管理，而编译器或程序员在“美学方面”做出的规划，本质上它并不是 CPU 用于内存访问的段，但它们都是描述了一段内存，而且程序中的段，其起始地址和大小可以理解为 CPU 访问内存分段策略中的“段基址：段内偏移地址”，这么说来，至少它们很接近了，让我们更近一步：程序是可以被人为划分成段的，并且可以将划分出来的段地址加载到段寄存器中，见下面的代码 0-1。

代码 0-1 程序分段

```
 1 section my_code vstart=0
 2   ;通过远跳转的方式给代码段寄存器 CS 赋值 0x90
 3     jmp 0x90:start
 4     start:      ;标号 start 只是为了 jmp 跳到下一条指令
 5
 6   ;初始化数据段寄存器 DS
 7     mov ax,section.my_data.start
 8     add ax,0x900   ;加 0x900 是因为本程序会被 mbr 加载到内存 0x900 处
 9     shr ax,4     ;提前右移 4 位,因为段基址会被 CPU 段部件左移 4 位
10     mov ds,ax
11
12   ;初始化栈段寄存器 SS
13     mov ax,section.my_stack.start
14     add ax,0x900  ;加 0x900 是因为本程序会被 mbr 加载到内存 0x900 处
15     shr ax,4    ;提前右移 4 位,因为段基址会被 CPU 段部件左移 4 位
16     mov ss,ax
17     mov sp,stack_top   ;初始化栈指针
18
19   ;此时 CS、DS、SS 段寄存器已经初始化完成,下面开始正式工作
20     push word [var2]   ;变量名 var2 编译后变成 0x4
21     jmp $
22
23   ;自定义的数据段
24 section my_data align=16 vstart=0
25     var1 dd 0x1
26     var2 dd 0x6
27
28   ;自定义的栈段
29 section my_stack align=16 vstart=0
30     times 128 db 0
31 stack_top:  ;此处用于栈顶,标号作用域是当前 section,
             ;以当前 section 的 vstart 为基数
32
```

代码 0-1 是实模式下运行的程序，其中自定义了三个段，为了和标准的段名（.code、.data 等）有所区别，这里代码段取名为 my_code，数据段取名为 my_data，栈段取名为 my_stack。这段代码是由 MBR 加载到物理内存地址 0x900 后，mbr 通过“jmp 0x900”跳过来的，我们的想法是让各段寄存器左移 4 位后的段基址与程序中各分段实际内存位置相同，所以对于代码段，希望其基址是 0x900，故代码段 CS 的值为 0x90（在实模式下，由 CPU 的段部件将其左移 4 位后变成 0x900，所以要初始化成左移 4 位前的值）。

但没有办法直接为 CS 寄存器赋值，所以在代码 0-1 开头，用“jmp 0x90：0”初始化了程序计数器 CS 和 IP。这样段寄存器 CS 就是程序中咱们自己划分的代码段了。

在此提醒一下，各 section 中的定义都有 align=16 和 vstart=0，这是用来指定各 section 按 16 位对齐的，各 section 的起始地址是 16 的整数倍，即用十六进制表示的话，最后一位是 0。所以右移操作如第 9 行的 shr ax，4，结果才是正确的，只是把 0 移出去了。否则不加 align=16 的话，section 的地址不能保证是 16 的整数倍，右移 4 位可能会丢数据。vstart=0 是指定各 section 内数据或指令的地址以 0 为起始编号，这样做为段内偏移地址时更方便。具体 vstart 内容请参阅本书相应章节。

第 6～10 行是初始化数据段寄存器 DS，是用程序中自己划分的段 my_data 的地址来初始化的。由于代码 0-1 本身是脱离操作系统的程序，是 MBR 将其加载到 0x900 后通过跳转指令“jmp 0x900”跳入执行的，所以要将 my_data 在文件内的地址 section.my_data.start 加上 0x900 才是最终在内存中的真实地址。右移 4 位的原因同代码段相同，都是 CPU 的段部件会自动将段基址左移 4 位，故提前右移 4 位。此地址作为段基址赋值给 DS，这样段寄存器 DS 中的值是程序中咱们自己划分的数据段了。

第 12～17 行是初始化栈段寄存器，原理和数据段差不多，唯一区别是栈段初始化多了个针指针 SP，为它初始化的值 stack_top 是最后一行，因为栈指针在使用过程中指向的地址越来越低，所以初始化时一定得是栈段的最高地址。

经过代码段、数据段、栈段的初始化，CPU 中的段寄存器 CS、DS、SS 都是指向程序中咱们自己划分的段地址，之后 CPU 的内存分段机制“段基址：段内偏移地址”，段基址就是程序中咱们自己划分的段，段内偏移地址都是各自定义段内的指令和数据地址，由于在 section 中有 vstart=0 限制，地址都是从 0 开始编号的。所以，程序中的分段和 CPU 内存访问的分段又是一回事。

让我们对此感到疑惑的原因，可能是我们一般都是用高级语言开发程序，在高级语言中，程序分段这种工作不由我们控制，是由编译器在编译阶段完成的。而且现代操作系统都是在平坦模型（整个 4GB 空间为 1 个段）下工作，编译器也是按照平坦模型为程序布局，程序中的代码和数据都在同一个段中整齐排列。大家可以用 readelf –e /bin/ls 查看一下 ls 命令，结果太长，就不截图啦。咱们主要关注三段内容。

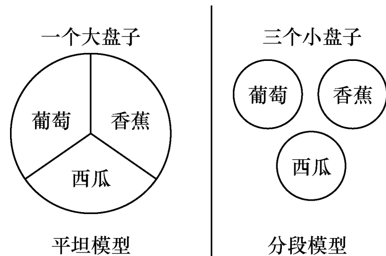
- Section Headers：列出了程序中所有的 section，这些 section 是 gcc 编译器帮忙划分的。
- Program Headers：列出了程序中的段，即 segment，这是程序中 section 合并后的结果。
- Section to Segment mapping：列出了一个 segment 中包含了哪些 section。

有关 section 和 segment 的内容请参见本书相关章节。

在 Section Headers 和 Program Headers 中您会发现，这些分段都是按照地址由低到高在 4GB 空间中连续整洁地分布的，在平坦模型下和谐融洽。

显然，不用程序员手工分段，并且采用平坦模型，这种操作上的“隔离”固然让我们更加方便，但也让我们更加感到进程空间布局的神秘。如果程序分段像代码 0-1 那样地直白、亲民，大家肯定不会感到迷惑了。其实我想说的是无论是否为平坦模型，程序中的分段和 CPU 中的内存分段机制，它们属于物品与容器的关系。

举个例子，程序中划分的段相当于各种水果，比如代码段相当于香蕉，数据段相当于葡萄，栈段相当于西瓜。CPU 内存分段策略中的段寄存器相当于盛水果的盘子。可以用一个大盘子将各种水果都放进来，但依然是分门别类地摆放，不能失去美感混成一锅粥，这就是段大小为 4GB 的平坦模型。也可以把每种水果分别放在一个小盘子里一块儿端上来，这就是普通的分段模型，如图 0-4 所示。



▲图 0-4　程序中分段在平坦模型和分段模型中的区别

总结一下，程序中的段只是逻辑上的划分，用于不同数据的归类，但是可以用 CPU 中的段寄存器直接指向它们，然后用内存分段机制去访问程序中的段，在这一点上看，它们很像相片和相框的关系：程序中的段是内存中的内容，相当于相片，属于被展示的内容，而内存分段机制则是访问内存的手段，相当于相框，有了相框，照片才能有地摆放。

我想大家应该已经搞清楚了内存分段和程序分段的关系，其实就是一回事，内存分段指的是处理器为访问内存而采用的机制，称之为内存分段机制，程序分段是软件中人为逻辑划分的内存区域，它本身也是内存，所以处理器在访问该区域时，也会采用内存分段机制，用段寄存器指向该区域的起始地址。

0.9 物理地址、逻辑地址、有效地址、线性地址、虚拟地址的区别

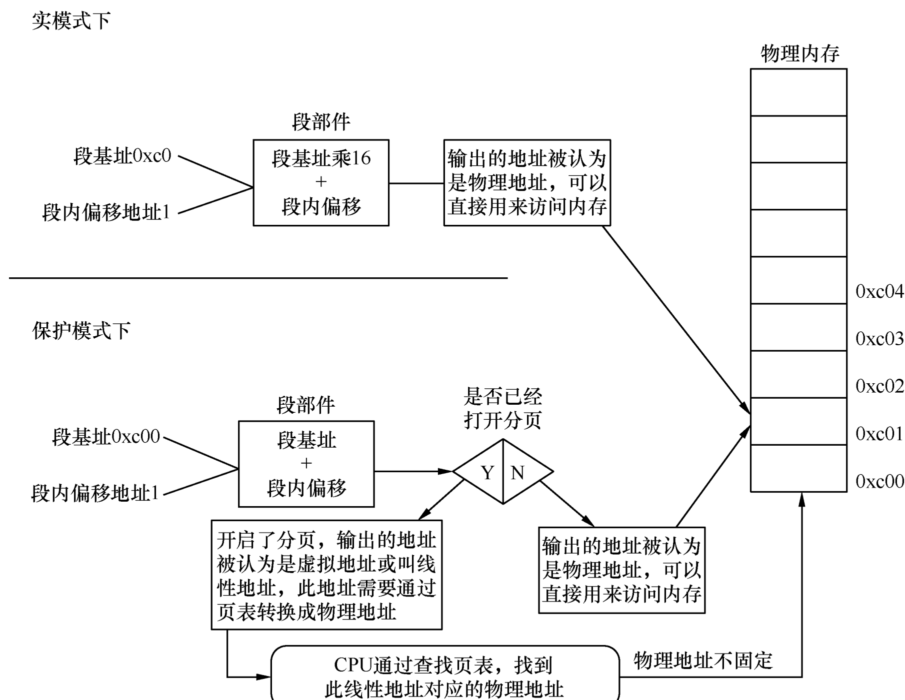
物理地址就是物理内存真正的地址，相当于内存中每个存储单元的门牌号，具有唯一性。不管在什么模式下，不管什么虚拟地址、线性地址，CPU 最终都要以物理地址去访问内存，只有物理地址才是内存访问的终点站。

在实模式下，“段基址+段内偏移地址”经过段部件的处理，直接输出的就是物理地址，CPU 可以直接用此地址访问内存。

而在保护模式下，“段基址+段内偏移地址”称为线性地址，不过，此时的段基址已经不再是真正的地址了，而是一个称为选择子的东西。它本质是个索引，类似于数组下标，通过这个索引便能在 GDT 中找到相应的段描述符，在该描述符中记录了该段的起始、大小等信息，这样便得到了段基址。若没有开启地址分页功能，此线性地址就被当作物理地址来用，可直接访问内存。若开启了分页功能，此线性地址又多了一个名字，就是虚拟地址（虚拟地址、线性地址在分页机制下都是一回事）。虚拟地址要经过 CPU 页部件转换成具体的物理地址，这样 CPU 才能将其送上地址总线去访问内存。

无论在实模式或是保护模式下，段内偏移地址又称为有效地址，也称为逻辑地址，这是程序员可见的地址。这是因为，最终的地址是由段基址和段内偏移地址组合而成的。由于段基址已经有默认的啦，要么是在实模式下的默认段寄存器中，要么是在保护模式下的默认段选择子寄存器指向的段描述符中，所以只要给出段内偏移地址就行了，这个地址虽然只是段内偏移，但加上默认的段基址，依然足够有效。

线性地址或称为虚拟地址，这都不是真实的内存地址。它们都用来描述程序或任务的地址空间。由于分页功能是需要在保护模式下开启的，32 位系统保护模式下的寻址空间是 4GB，所以虚拟地址或线性地址就是 0～4GB 的范围。转换过程如图 0-5 所示。

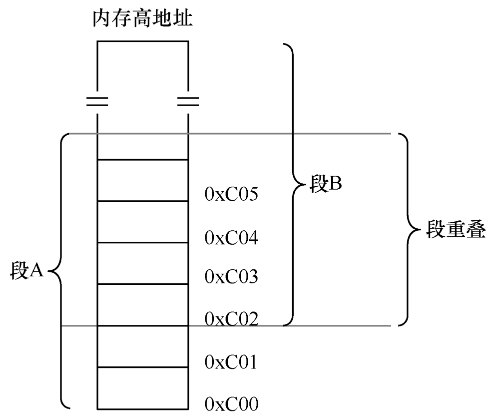


▲图 0-5　虚拟地址、物理地址等

0.10 什么是段重叠

其实上面已经提到了段重叠，也许有的读者已经明白了，但还是在此特意解释一下吧。

依然假设在实模式下（并不是说在保护模式下就不存在段重叠，只是这样就会少解释了相关数据结构，如段描述符，不过这不重要，原理是一样的），一个段最大为 64KB，其大小由段内偏移地址寻址范围决定，也就是 2 的 16 次方。其起始位置由段基地址决定。CPU 的内存寻址方式是：给我一个段基址，再给我一个相对于该段起始位置的偏移地址，我就能访问到相应内存。它并不要求一个内存地址只隶属于某一个段，所以在上面的图 0-2 中，欲访问内存 0xC03，段基址可以选择 0xC00，0xC01，0xC02，0xC03，只不过是段内偏移量要根据段基地址来调整罢了。用这种“段基地址：段内偏移”的组合，0xC00：3 和 0xC02：1 是等价的，它们都访问到同一个物理内存块。但段的大小决定于段内偏移地址寻址范围，假设段 A 的段基址是从 0xC00 开始，段 B 的段基址是从 0xC02 开始，在 16 位宽度的寻址范围内，这两个段都能访问到 0xC05 这块内存。用段 A 去访问，其偏移为 5，用段 B 去访问，其偏移量为 3。这样一来，用段 B 和段 A 在地址 0xC02 之后，一直到段 B 偏移地址为 0xfffe 的部分，像是重叠在一起了，这就是段重叠了，如图 0-6 所示。



▲图 0-6　段重叠

0.11 什么是平坦模型

平坦模型是相对于多段模型来说的，所以说平坦模型指的就是一个段。比如在实模式下，访问超过 64KB 的内存，需要重新指定不同的段基址，通过这种迂回变通的方式才能达到目的。在保护模式下，由于其是 32 位的，寻址范围便能够达到 4GB，段内偏移地址也是地址，所以也是 32 位。可见，在 32 位环境下用一个段就能够访问到硬件所支持的所有内存。也就是说，段的大小可以是地址总线能够到达的范围。既然平坦模型是相对于多段模型来说的，为什么不称为单段模型，而称为平坦呢，我估计很多读者已经明白了，用多个小段再加上不断换段基址的方式访问内存确实够麻烦的，可能换着换着就晕了，别忘记了，这种多段模型为了访问到 1MB 地址空间，还需要额外打开 A20 地址线呢，这种访存方式本身就是种补救措施，相当于给硬件打了个补丁，既然是补丁，访问内存的过程必然是不顺畅的。相对于那么麻烦的多段模型，平坦模型不需要额外打开 A20 地址线，不需要来回切换段基址就可以在地址空间内任意翱翔。如果把内存段比喻成小格子的话，平坦模型下的内存访问，没有众多小格子成为羁绊，可谓一路“平坦”。

所以“平坦”这两个字，突显了当时的程序员受多段模型折磨之苦，迫不及待地想表达其优势的喜悦之情。

0.12 cs、ds 这类 sreg 段寄存器，位宽是多少

CPU 中存在段寄存器是因为其内存是分段访问的，这是设计之初决定的，属于基因里的东西。前面已经介绍过了内存分段访问的方法，这里不再赘述。

CPU 内部的段寄存器（Segment reg）如下。

（1）CS——代码段寄存器（Code Segment Register），其值为代码段的段基值。

（2）DS——数据段寄存器（Data Segment Register），其值为数据段的段基值。

（3）ES——附加段寄存器（Extra Segment Register），其值为附加数据段的段基值，称为“附加”是因为此段寄存器用途不像其他 sreg 那样固定，可以额外做他用。

（4）FS——附加段寄存器（Extra Segment Register），其值为附加数据段的段基值，同上，用途不固定，使用上灵活机动。

（5）GS——附加段寄存器（Extra Segment Register），其值为附加数据段的段基值。

（6）SS——堆栈段寄存器（Stack Segment Register），其值为堆栈段的段值。

32 位 CPU 有两种不同的工作模式：实模式和保护模式。

每种模式下，段寄存器中值的意义是不同的，但不管其为何值，在段寄存器中所表达的都是指向的段在哪里。在实模式下，CS、DS、ES、SS 中的值为段基址，是具体的物理地址，内存单元的逻辑地址仍为“段基值：段内偏移量”的形式。在保护模式下，装入段寄存器的不再是段地址，而是“段选择子”（Selector），当然，选择子也是数值，其依然为 16 位宽度。

可见，在 32 位 CPU 中，sreg 无论是工作在 16 位的实模式，还是 32 位的保护模式，用的段寄存器都是同一组，并且在 32 位下的段选择子是 16 位宽度，排除了段寄存器在 32 位环境下是 32 位宽的可能，综上所述，sreg 都是 16 位宽。

0.13 什么是工程，什么是协议

这两个小问题，一些非开发型技术人员经常会问到，做过开发的同学肯定了解。想想还是简单说一下吧（因为这名词似乎也没法说复杂）。

软件中的工程是指开发一套软件所需要的全部文件，包括配置环境。

在一般的集成开发环境中如 eclipse 或 vc++，在程序的开始都是先建立一个 project，这就是所谓的工程，它相当于一个大目录，以后写的代码都在这里面。

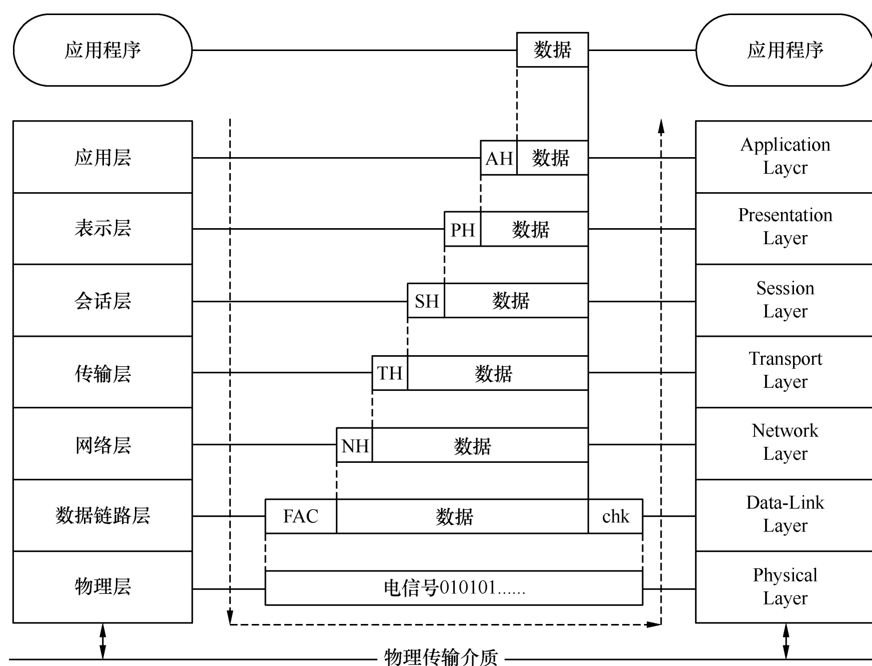
全部文件包含实际代码和环境配置两部分。实际代码部分，除了自己写的代码文件之外，一般都要包含其他同事写的头文件，若是与他方合作，还要包含第三方头文件。环境配置部分，一般是配置一些模板、库文件目录，具体还要根据所用的实际框架来配置，包含一些服务器的地址，端口之类也都在配置文件中。还是那句话，工程就是为了完成软件编写所涉及的全部相关文件。

协议是一种大家共同遵守的规约，主要用来实现通信、共享、协作；起初是为避免大家各干各的，无法彼此调用对方成果的情况，从而给大家统一一种接口、一组数据调用或者分析的约定。

大家达成一致后，都遵守这个约定开发自己的产品，别人只要也按照这个约定就能够享用自己的成果，从而实现了彼此兼容。只要是技术人员都对 TCP/IP 有所了解，这就是我们目前赖以生存的网络协议。根据 OSI 七层模型，它规定数据的第一层，也就是最外层物理层，这一层包含的是电路相关的数据。发送方和接收方都彼此认同最外层的就是电路传输用的数据。每一层中的前几个固定的字节必须是描述当前层的属性，根据此属性就能找到需要的数据。各层中的数据部分都是更上一层的数据，如第一层（物理层）中的数据部分是第二层（数据链路层）的属性+数据，第三层（网络层）的数据部分是第四层（传输层）TCP 或 UDP 的属性+数据。各层都是如此，直到第七层（应用层）的数据部分才是真正应用软件所需要的数据。由此可见，对方一大串数据发过来后，经过层层剥离处理，到了最终的接收方（应用软件），只是一小点啦。

如图 0-7 所示，两边的应用程序互发数据时，其实发的就是最顶层的那一小点“数据”，每下一层，便加了各层的报文头，上层整个（包括自己的报文头和报文体）全部成了下一层的数据部分。

这样说似乎还是很抽象，具体地说，就是需要的数据是在偏移文件固定大小的字节处，这个固定字节是多少，就是协议中所规定的。不了解 TCP/IP 的同学可以参看各层报文格式，自行查阅吧。



▲图 0-7　OSI 七层模型

0.14 为什么 Linux 系统下的应用程序不能在 Windows 系统下运行

其实，Windows 下的程序也无法直接在 Linux 下运行。

对于这个问题，很多同学都会马上给出答案：格式不同。其实……答对啦，确实是格式不同，不过这只是一方面，还有另一方面，系统 API 不同，API 即 Application Programming Interface，应用程序编程接口。

先说说格式。其实格式也算是协议，就是在某个固定的位置有固定意义的数据。Linux 下的可执行程序格式是 elf，也就是“Executable and Linking Format”平时咱们用 readelf 命令可以查看 elf 文件头，里面有节（section）信息、段（segment）信息、程序入口（entry_point）、哪个段由哪些节组成等信息。

而 Windows 下的可执行程序是 PE 格式（portable executable，可移植的可执行文件），因为我没了解过，所以具体文件头咱们就不关注了，有兴趣的同学自行查看。

那如果 Linux 支持了 PE 格式就可以运行 Wndows 程序了吗？也不行，因为在上面说过了，还有系统 API 不同。Linux 中的 API 称为系统调用，是通过 int 0x80 这个软中断实现的。而 Windows 中的 API 是存放在动态链接库文件中的，也就是 Windows 开发人员常说的 DLL，即 Dynamic Link Library 的缩写。LL 是一个库，里面包含代码和数据，可供用户程序调用，DLL 不是可执行文件，不能够单独运行。也就是说，Linux 中的可执行程序获得系统资源的方法和 Windows 不一样，所以显然是不能在 Windows 中运行的。

除以上原因外，这还和编译器、标准库有关，不再列举。

0.15 局部变量和函数参数为什么要放在栈中

局部变量，顾名思义其作用域属于局部，并不是像 static 那样属于全局性的。全局的变量，意味着谁都可以随时随地访问，所以其放在数据段中。而局部变量只是自己在用，放在数据段中纯属浪费空间，没有必要，故将其放在自己的栈中，随时可以清理，真正体现了局部的意义。这个就是堆栈框架，提到了就说一点吧，栈由于是向下生长的，堆栈框架就是把 esp 指针提前加一个数，原 esp 指针到新 esp 指针之间的栈空间用来存储局部变量。解释一个概念，堆是程序运行过程中用于动态内存分配的内存空间，是操作系统为每个用户进程规划的，属于软件范畴。栈是处理器运行必备的内存空间，是硬件必需的，但又是由软件（操作系统）提供的。

堆是堆，而堆栈就是栈，和堆没关系，只是都这么叫。栈和堆栈都是指的栈，在 C 程序的内存布局中，由于堆和栈的地址空间是接壤的，栈从高地址往低地址发展，堆是从低地址往高地址发展，堆和栈早晚会碰头，它们各自的大小取决于实际的使用情况，界限并不明朗，所以这可能是堆栈常放在一直称呼的原因吧。

函数参数为什么会放到栈区呢？第一也是其局部性导致的，只有这个函数用这个参数，何必将其放在数据段呢。第二，这是因为函数是在程序执行过程中调用的，属于动态的调用，编译时无法预测会何时调用及被调用的次数，函数的参数及返回值都需要内存来存储，如果是递归调用的话，参数及返回值需要的内存空间也就不确定了，这取决于递归的次数。也许这么说您也依然觉得费解，如果完全明白，需要了解一下编译原理，很多知识都是通过实践后才搞明白的。当然我不是说让您为了搞明白这个问题而去尝试写个编译器。

总之，在函数的编译阶段根本无法确定它会被调用几次，其参数和函数的返回地址也要内存来存储，所以也不知道其会需要多少内存。我想，即使神通广大的编译器设计者可以预测这些了，那提前准备好内存也是一种浪费，而且您想啊，在系统中可用内存紧缺的情况下，提前把内存分配给目前并不使用内存的进程（只因为要存储其函数参数），而眼前需要内存的程序若无内存可用，引用罗永浩老师的一句话："我想不到比这个更伤感的事情了"所以编译器为了让世界更美好一些，选择将为函数参数动态分配内存，也就是在每次调用函数时才为它在栈中分配内存。

0.16 为什么说汇编语言比 C 语言快

首先说这是谬论（有没有想喷我的冲动？大人且慢，请听我慢慢道来）。

不管用什么语言，程序最终都是给 CPU 运行的，只有 CPU 才能让程序跑起来。CPU 不知道什么是汇编语言、C 语言，甚至 Java、PHP、Python 等，它根本不知道交给它的指令曾经经历过那么多的解释、编译工序。不管什么语言，编译器最终翻译出来的都是机器指令。所以在这一点来说，汇编语言编译器编译出来的机器指令和 C 编译器编译出来的机器指令无异。

那为什么还说汇编语言更快呢？

我觉得应该说汇编语言生成的指令数更少，从而"显得"执行得快，并不是汇编语言本身有多少威武霸气，而是因为汇编语言本身就是机器指令的符号化，意思是说，一个汇编语言中的符号对应一个机器指令，它们是一一对应的。用汇编语言写程序就相当于直接在写机器指令，汇编语言编译器并不会添加额外的语句，因此汇编语言写的程序会更直接，CPU 不会因多执行一些无关的指令而浪费时间，当然会快。

再看看 C 编译器为咱们做了什么。为了让 C 程序员更加方便地编程，C 编译器在背后做了大量的工作，不仅如此，出于通用性、易用性或者其他方面的考虑，C 编译器往往会在背后加入额外的 C 语言代码来支撑，因此实际的 C 代码量就变得很大。另外在编译阶段，C 代码会率先被编译成汇编代码，然后再由汇编器将汇编代码翻译成机器指令，由于 C 代码已经变得冗余了，编译出的汇编代码自然也会冗余，其机器指令也会多很多。

大多数人愿意用 C 语言写程序是因为 C 语言强大且更容易掌握。但这份优势是有代价的。C 程序员不用考虑切换栈，不用考虑用哪个段。这些必须要考虑的事情，程序员不考虑，只好由编译器帮着考虑了。而且为了通用性、功能，甚至安全方面的考虑，自然在背后要多写一些代码。就拿打印字符串来说，C 语言的 printf()，这里面的工作可多了去了，不仅要检查打印的数据类型，还要负责格式，小数点保留位数……而在汇编语言中只要往显存地址处 mov 一个字符就行了，字符串也就是多几个 mov 操作而已。您说，C 语言为了让开发者用得爽，自己在背后做了多少贡献。

总结：高级语言如 C 语言为了通用性等，需要兼顾的东西比较多，往往还加入了一些额外的代码，因此编译出来的汇编代码比较多，很多部分都是一些周边功能，并不是直接起作用的，不如用汇编语言直接写功能相关的部分效果来得更直接，C 语言被编译成机器指令后，生成的机器指令当然也包括这些额外的部分，相当于多执行了一些"看似没用"的指令，因此会比直接用汇编语言慢。

0.17 先有的语言，还是先有的编译器，第 1 个编译器是怎么产生的

首先肯定的是先有的编程语言，哪怕这个语言简单到只有一个符号。先是设计好语言的规则，然后编写能够识别这套规则的编译器，否则若没有语言规则作为指导方向，编译器编写将无从下笔。

第 1 个编译器是怎么产生的？这个问题我并没有求证，不过可以谈下自己的理解，请大伙儿辩证地看。

这个问题属于哲学中鸡生蛋、蛋生鸡的问题，这种思维回旋性质的本源问题经常让人产生迷惑。可是现实生活中这样的例子太多了。

（1）英语老师教学生英语，学生成了英语老师后又可以教其他学生英语。

（2）写新的书需要参考其他旧书，新的书将来又会被更新的书参考，就像本书编写过程一样，要参考许多前辈的著作。

（3）用工具可以制造工具，被制造出来的工具将来又可以制造新的工具。

（4）编译器可以编译出新的编译器。

这种自己创造自己的现象，称为自举。

自举？是不是自己把自己举起来？是的，人是不能把自己举起来的，这个词很形象地描述了这类“后果必须有前因”的现象。

以上前三个列举的都是生活例子，似乎比第 4 个更容易接受。即使这样，对于前三个例子大家依然会有疑问。

（1）第一个会英语的人是谁教的？

（2）第一本书是怎样产生的？

（3）第一个工具是如何制造出来的？

其实看到第 2 个例子大家就可能明白了，世界上的第一本书，它的知识来源肯定是人的记忆，通过向个人或群众打听，把大家都认同的知识记录到某个介质上，这样第一本书就出生了。此后再记录新的知识时，由于有了这本书的参考，不需要重新再向众人打听原有知识了，从此以后便形成了书生书的因果循环。

从书的例子可以证明，本源问题中的第一个，都是由其他事物创建出来的，不是自己创造的自己。

就像先有鸡还是先有蛋一样，一定是先有其他生命体，这个生命体不是今天所说的鸡。伴随这个生命体漫长的进化中，突然有一天它具备了生蛋的能力（也许这个蛋在最初并不能孵化成鸡，这个生命体又经过漫长的进化，最终可以生出能够孵化成鸡的蛋），于是这个蛋可以生出鸡了。过了很久之后，才有的人类。人一开始接触的便是现在的鸡而不知道那个生命体的存在，所以人只知道鸡是由蛋生出来的。

很容易让人混淆的是编译 C 语言时，它先是被编译成汇编代码，再由汇编代码编译为机器码，这样很容易让人误以为一种语言是基于一种更底层的语言。

似乎没有汇编语言，C 语言就没有办法编译一样。拿 gcc 来说，其内部确实要调用汇编器来完成汇编语言到机器码的翻译工作。因为已经有了汇编语言编译器，那何必浪费这个资源不用，自己非要把 C 语言直接翻译成机器码呢，毕竟汇编器已经无比健壮了，将 C 直接变成机器码这个难度比将 C 语言翻译为汇编语言大多了，这属于重新造轮子的行为。

曾经我就这样问过自己，PHP 解释器是 C 语言写的，C 编译器是汇编写的（这句话不正确），汇编是谁写的呢？后来才知道，编译器 GCC 其实是用 C 语言写的。乍一听，什么？用 C 语言写 C 编译器？自己创造自己，就像电影超验骇客一样。当时的思维似乎陷入了死循环一样，现在看来这不奇怪。其实编译器用什么语言写是无所谓的，关键是能编译出指令就行了。编译出的可执行文件是要写到磁盘上的，理论上，只要某个进程，无论其是不是编译器，只要其关于读写文件的功能足够强大，可以往磁盘上写任意内容，都可以生成可执行文件，直接让操作系统加载运行。想象一下，用 Python 写一个脚本，功能是复制一个二进制可执行文件，新复制出来的文件肯定是可以执行的。那 Python 脚本直接输出这样的一个二进制可执行文件，它自然就是可以直接执行的，完全脱离 Python 解释器了。

编译器其实就是语言，因为编译器在设计之初就是先要规划好某种语言，根据这个语言规则来写合适的编译器。所以说，要发明一种语言，关键是得写出与之配套的编译器，这两者是同时出来的。最初的编译器肯定

是简单粗糙的，因为当时的编程语言肯定不完善，顶多是几个符号而已，所以难以称之为语言。只有功能完善且符合规范，有自己一套体系后才能称之为语言。不用说，这个最初的编译器肯定无法编译今天的 C 语言代码。编程语言只是文本，文本只是用来看的，没有执行能力。最初的编译器肯定是用机器码写出来的。这个编译器能识别文本，可以处理一些符号关键字。随着符号越来越多，不断地改进这个编译器就是了。

以上的符号就是编程语言。后来这个编译器支持的关键字越来越多了，也就是这个编译器支持的编程语言越发强大了，可以写出一些复杂的功能的时候，干脆直接用这个语言写个新的编译器，这个新的编译器出生时，还是需要用老的编译器编译出来的。只要有了新的编译器，之后就可以和老的编译器说拜拜了。发明新的编译器实际上就是为了能够处理更多的符号关键字，也就是又有新的开发语言了，这个语言可以是全新的，也可以是最初的语言，这取决于编译器的实现。这个过程不断持续，不断进化，逐渐才有了今天的各种语言解释器，这是个迭代的过程。



▲图 0-8　雕刻（来源网络）

图 0-8 所示这张图片在网络上非常火，它常常与励志类的文字相关。起初看到这个雕像在雕刻自己时，我着实被感动了，感受到的是一种成长之痛。今天把它贴过来的目的是想告诉大家，起初的编译器也是功能简单，不成规范，然而经过不断自我“雕刻”，它才有了今天功能的完善。

下面的内容我参考了别人的文章，由于找不到这位大师的署名，只好在此先献上我真挚的敬意，感谢他对求知者的奉献。

要说到 C 编译器的发展，必须要提到这两位大神——C 语言之父 Dennis Ritchie 和 Ken Thompson。Dennis 和 Ken 在编程语言和操作系统的深远贡献让他们获得了计算机科学的最高荣誉——Dennis 和 Ken 于 1983 年赢得了 ACM 图灵奖。

编译器是靠不断学习、积累才发展起来的，这是自我学习的过程，下面来看看他们是如何让编译器长大的。

起初的 C 编译器中并没有处理转义字符，为叙述方便，我们现在称之为老编译器。如果待编译的代码文件中有字符串'\\'，在老编译器眼里，这就是'\\'字符串，并不是转义后的单个字符'\'。为了表明编译器与作为其输入的代码文件的关系，我们称作为输入的代码文件为应用程序文件，毕竟虽然待编译的代码文件实现了一个编译器，但在编译器眼里，它只是一个应用程序级角色。例如，gcc –c a.c 中，a.c 就是应用程序文件。

现在想在编译器中添加对转义字符的支持，那就需要修改老编译器的源代码，假设老编译器的源代码文件名为 compile_old.c。被修改后的编译器代码，已不属于老编译器的源代码，故我们命名其文件名为 compile_new_a.c，下面是修改后的内容。

代码 compile_new_a.c

```
1  ...
2  c = next();
3  if(c != '\')
4  return c;
5  c = next();
6  if(c == '\')
7  return '\';
8  ...
```

用老编译器将新编译器的源代码 compile_new_a.c 编译，生成可执行文件，该文件就是新的编译器，我们取名为新编译器_a。为了方便理清它们的关系，将它们列入表格中。

编译器自身源代码	编译器	应用程序源代码	输出文件名
compile_old.c	老编译器	compile_new_a.c	新编译器_a，支持'\\'

这下编译出来的新编译器_a 可以编译含有转义字符'\'的应用程序代码了，也就是说，待编译的文件（也就是应用程序代码）中，应该用'\\'来表示'\'。而单独的字符'\'在新编译器_a 中未做处理而无法通过编译。所以此时新编译器_a 是无法编译自己的源代码 compile_new_a.c 的，因为该源文件中只是单个'\'字符，新编译器_a 只认得'\\'。

先更新它们的关系，见下表。

编译器自身源代码	编译器	应用程序源代码	输出文件名
compile_old.c	老编译器	compile_new_a.c	新编译器_a，支持'\\'
compile_new_a.c	新编译器_a	compile_new_a.c	编译失败

也就是说，现在新编译器_a 无法编译自己的源文件 compile_new_a.c，只有老编译器才能编译它。

分析一下，新编译器_a 无法正确编译自己的源文件 compile_new_a.c，其原因是 compile_new_a.c 中'\'字符应该用转义字符的方式来引用，即所有用'\'的地方都应该替换为'\\'。再啰嗦一下，请见新编译器_a 的源代码 compile_new_a.c，它只处理了字符串'\\'，单个'\'没有对应的处理逻辑。下面修改代码，将新修改后的代码命名为 compile_new_b.c。

代码 compile_new_b.c

```
1  ...
2  c = next();
3  if(c != '\\')
4  return c;
5  c = next();
6  if(c == '\\')
7  return '\\';
8  ...
```

其实 compile_new_b.c 只是更新了转义字符的语法，这是新编译器_a 所支持的新的语法，此文件是否是编译器源码没什么关系。所以下面还是以新编译器_a 来编译新的编译器。

用新编译器_a 编译此文件，将生成新编译器_b，将新的关系录入到表格中。

编译器自身源代码	编译器	应用程序源代码	输出文件名
compile_old.c	老编译器	compile_new_a.c	新编译器_a，支持'\\'
compile_new_a.c	新编译器_a	compile_new_a.c	编译失败
compile_new_a.c	新编译器_a	compile_new_b.c	新编译器_b，支持'\'

现在想加上换行符'\n'的支持。

```
1  if(c == 'n')
2  return '\n';
```

由于现在编译器还不认识'\n'，故这样做肯定不行，不过可以用其 ASCII 码来代替，将其命名为 compile_new_c.c。

代码 compile_new_c.c

```
1   ...
2   c = next();
3   if(c != '\')
4   return c;
5   c = next();
6   if(c == '\')
7   return '\' ;
8   if(c == 'n')
9   return 10;
10  ...
```

用新编译器_a 来编译 compile_new_c.c，将生成新编译器_c。

编译器自身源代码	编译器	应用程序源代码	输出文件名
compile_old.c	老编译器	compile_new_a.c	新编译器_a，支持'\\'
compile_new_a.c	新编译器_a	compile_new_a.c	编译失败
compile_new_a.c	新编译器_a	compile_new_b.c	新编译器_b，支持'\'
compile_new_a.c	新编译器_a	compile_new_c.c	新编译器_c，间接支持“\n”

最后再修改 compile_new_c.c 为 compile_new_d.c，将 10 用'\n'替代。

代码 compile_new_d.c

```
1  ...
2  c = next();
3  if(c != '\\')
4  return c;
5  c = next();
6  if(c == '\\')
7  return '\\';
8  if(c == 'n')
9  return '\n';
10 ...
```

用新编译器_c 编译 compile_new_d.c，生成新编译器 d，将直接识别'\n'。

编译器自身源代码	编译器	应用程序源代码	输出文件名
compile_old.c	老编译器	compile_new_a.c	新编译器_a，支持'\\'
compile_new_a.c	新编译器_a	compile_new_a.c	编译失败
compile_new_a.c	新编译器_a	compile_new_b.c	新编译器_b，支持'\'
compile_new_a.c	新编译器_a	compile_new_c.c	新编译器_c，间接支持“\n”
compile_new_c.c	新编译器_c	compile_new_d.c	新编译器 d，直接支持“\n”

编译器经过这样不断的训练，功能越来越强大，不过体积也越来越大了。

0.18 编译型程序与解释型程序的区别

解释型语言，也称为脚本语言，如 JavaScript、Python、Perl、PHP、Shell 脚本等。它们本身是文本文件，是某个应用程序的输入，这个应用程序是脚本解释器。

由于只是文本，这些脚本中的代码在脚本解释器看来和字符串无异。也就是说，脚本中的代码从来没真正上过 CPU 去执行，CPU 的 cs：ip 寄存器从来没指向过它们，在 CPU 眼里只看得到脚本解释器，而这些脚本中的代码，CPU 从来就不知道有它们的存在。这些脚本代码看似在按照开发人员的逻辑执行，本质上是脚本解释器在时时分析这个脚本，动态根据关键字和语法来做出相应的行为。因此脚本中若出现错误，先前正确的部分也会被正常执行，这和编译型程序有很大区别。

顺便猜想一下解释型语言是如何执行的。我们在执行一个 PHP 脚本时，其实就是启动一个 C 语言编写出来的解释器而已，这个解释器就是一个进程，和一般的进程是没有区别的，只是这个进程的输入则是这个 php 脚本，在 php 解释器中，这个脚本就是个长一些的字符串，根本不是什么指令代码之类。只是这种解释器了解这种语法，按照语法规则来输出罢了。

举个例子，假设下面是文件名为 a.php 的 PHP 代码。

```
<?php                    这是 php 语法中的固定开始标签
    echo "abcd";         输出字符串 abcd
?>                       固定结束标签
```

PHP 解释器分析文本文件 a.php 时，发现里面的 echo 关键字，将其后面的参数获取后就调用 C 语言中提供的输出函数，如 printf（(echo 的参数)）。PHP 解释器对于 PHP 脚本，就相当于浏览器对于 JavaScript 一样，不过这个可完全是我猜测的，我不知道 PHP 解释器里面的具体工作，以上为了说清楚我的想法，请大家辩证地看。

而编译型语言编译出来的程序，运行时本身就是一个进程。它是由操作系统直接调用的。也就是由操作系统加载到内存后，操作系统将 CS：IP 寄存器指向这个程序的入口，使它直接上 CPU 运行。总之调度器在就绪队列中能看到此进程。而解释型程序是无法让调度器“入眼”的，调度器只会看到该脚本语言的解释器。

0.19 什么是大端字节序、小端字节序

先说一下为什么会产生字节序的问题。

内存是以字节为单位读写的，其最小的读写单位就是字节。故如果在内存中只写入一个字节，一个内存的存储单元便可将其容纳了，只要访问这一内存地址就能够完整取出这 1 字节。可是 1 字节要能够表示的范围只有 0～255（先只考虑无符号数），超过这个范围的数，只好用多个字节连在一起来表示。因此，在我们的 32 位程序中，定义的数据类型很多。1 字节的数据类型只有 char 型，像 int 型要占 4 字节，double 型要占用 8 字节。正如解决了一个问题又抛出了新的问题一样，解决了数值范围的问题，那带来的新的问题是这么多个字节该以怎样的顺序排放呢。一个超过 255 的数字必然要占用 2 个字节以上，这两个字节，在物理内存中，哪个在前？哪个在后？拿 0x1234 举例，数值中的高位 12 是放在内存的高地址处，还是低地址处？

于是就产生了这两种相反的排列顺序。

（1）小端字节序是数值的低字节放在内存的低地址处，数值的高字节放在内存的高地址。

（2）大端字节序是数值的低字节放在内存的高地址处，数值的高字节放在内存的低地址。

为了让大家理解得更直观，我在虚拟机 bochs 中操作一下，咱们看一下真正的 0x12345678 在内存中是怎样存储的，如图 0-9 所示。

上面的 b 0x7c00 是我在内存的 0x7c00 处插入了一个断点，其实这与要说明的问题无关，怕有同学好奇就稍带说一句，因为 0x7c00 是 BIOS 把 mbr 加载到内存后会跳转过去的地址，所以在此处能停下来。咱们只要关注 xp/4 0x200000，这是显示以物理内存 0x200000 开始处的 4 个字节，可见其为 00、00、00、00，地址是从左到右逐渐升高的，其中每一对 00 就占用 1 个字节，它们的值都是 0。现在用 setpmem 命令在该地址处写入 0x12345678 后，再用 xp/4 命令查看内存地址 0x200000 处的内容，可见已经不是 4 个 00 了，由内存的低地址到高地址，依次变成了 0x78、0x56、0x34、0x12。这说明 bochs 模拟的 x86 体系结构虚拟机是小端字节序，即数值上的低字节 0x78 在物理内存上的低地址，其他数值也依次符合小端字节序。

选择哪种字节序，这是硬件厂商考虑的问题，对于这种二选一的选择，选择了一方的时候，就必然丢了另一方。

看看这两种字节序的优势。

（1）小端：因为低位在低字节，强制转换数据型时不需要再调整字节了。

（2）大端：有符号数，其字节最高位不仅表示数值本身，还起到了符号的作用。符号位固定为第一字节，也就是最高位占据最低地址，符号直接可以取出来，容易判断正负。

简要说明一下小端的优势。因为在做强制数据类型转换时，如果转换是由低精度转向高精度，这数值本身没什么变化，如 short 是 2 字节，将其转换为 4 字节的 int 类型，无非是由 0x1234 变成了 0x00001234，数值上是不变的，只是存储形式上变了。如果转换是高精度转向低精度，也就是多个字节的数值要减少一些存储字节，这必然是要丢弃一部分数值。编译器的转换原则是强制转换到低精度类型，丢弃数值的高字节位，只保留数值的低字节，如图 0-10 所示。

```
<bochs:1> b 0x7c00
<bochs:2> xp/b
[bochs]:
0x00000000 <bogus+       0>:    0x00
<bochs:3> xp/4 0x200000
[bochs]:
0x00200000 <bogus+       0>:    0x00    0x00    0x00    0x00
<bochs:4> setpmem 0x200000 4 0x12345678
<bochs:5> xp/4 0x200000
[bochs]:
0x00200000 <bogus+       0>:    0x78    0x56    0x34    0x12
<bochs:6>
```

▲图 0-9　内存中存储形式

```
[work@localhost tmp]$ cat 1.c
#include <stdio.h>
int main() {
   unsigned int a = 0x12345678;
   printf("int %x, short %x\n", a, (short)a);
   return 0;
}
[work@localhost tmp]$ ./1
int 12345678, short 5678
[work@localhost tmp]$
```

▲图 0-10　强制类型转换与字节序

由图 0-10 上输出可见，0x12345678 由 4 字节的 int 型强制转向了 2 字节的 short 型后，只保留了低字节的 0x5678。

对于大端的优势，就硬件而言，就是符号位的判断变得方便了。最高位在最低地址，也就是直接就可以取到了，不用再跨越几个字节，减少了时钟周期。另外，对于人类来说，还是大端看上去顺眼，毕竟咱们存储 0x12345678 到内存时，它在内存中的存储顺序也是 0x12345678，而不是 0x78563412，这样看上去才直观。

常见 CPU 的字节序如下。

（1）大端字节序：IBM、Sun、PowerPC。

（2）小端字节序：x86、DEC。

ARM 体系的 CPU 则大小端字节序通吃，具体用哪类字节序由硬件选择。

字节序不仅是在 CPU 访问内存中的概念，而且也包括在文件存储和网络传输中。bmp 格式的图片就属于小端字节序，而 jpeg 格式的图片则为大端字节序，这没什么可说的，采用什么序列完全是开发者设计产品时的需要。

网络字节序就是大端字节序，所以在 x86 架构上的程序在发送网络数据时，要转换字节顺序。

关于字节序就介绍到这里，读者若觉得意犹未尽可以自行查阅。

0.20 BIOS 中断、DOS 中断、Linux 中断的区别

在计算机系统中，无论是在实模式，还是在保护模式，在任何情况下都会有来自外部或内部的事件发生。如果事件来自于 CPU 内部就称为异常，即 Exception。例如，CPU 在计算算法时，发现分母为 0，就抛出了除 0 异常。如果事件来自于外部，也就是该事件由外部设备发出并通知了 CPU，这个事件就称为异常。

BIOS 和 DOS 都是存在于实模式下的程序，由它们建立的中断调用都是建立在中断向量表（Interrupt Vector Table，IVT）中的。它们都是通过软中断指令 int 中断号来调用的。

中断向量表中的每个中断向量大小是 4 字节。这 4 字节描述了一个中断处理例程（程序）的段基址和段内偏移地址。因为中断向量表的长度为 1024 字节，故该表最多容纳 256 个中断向量处理程序。计算机启动之初，中断向量表中的中断例程是由 BIOS 建立的，它从物理内存地址 0x0000 处初始化并在中断向量表中添加各种处理例程。

BIOS 中断调用的主要功能是提供了硬件访问的方法，该方法使对硬件的操作变得简单易行。这句话是否也表明了不通过 BIOS 调用也是可以访问硬件的？必须是的，否则 BIOS 中断处理程序又是如何操作硬件呢？操作硬件无非是通过 in/out 指令来读写外设的端口，BIOS 中断程序处理是用来操作硬件的，故该处理程序中一定到处都是 in/out 指令。

BIOS 为什么添加中断处理例程呢？

（1）给自己用，因为 BIOS 也是一段程序，是程序就很可能要重复性地执行某段代码，它直接将其写成中断函数，直接调用多省心。

（2）给后来的程序用，如加载器或 boot loader。它们在调用硬件资源时就不需要自己重写代码了。

BIOS 是如何设置中断处理程序的呢？

BIOS 也要调用别人的函数例程。

BIOS 够底层吧？难道它还要依赖别人？是啊，BIOS 也是软件，也要有求于别人。首先硬件厂商为了让自己生产的产品易用，肯定事先写好了一组调用接口，必然是越简单越好，直接给接口函数传一个参数，硬件就能返回一个输出，如果不易用的话，厂商肯定倒闭了。

那这些硬件自己的接口代码在哪里呢？

每个外设，包括显卡、键盘、各种控制器等，都有自己的内存（主板也有自己的内存，BIOS 就存放在里面），不过这种内存都是只读存储器 ROM。硬件自己的功能调用例程及初始化代码就存放在这 ROM 中。根据规范，第 1 个内存单元的内容是 0x55，第 2 个存储单元是 0xAA，第 3 个存储单位是该 rom 中以 512 字节为单位的代码长度。从第 4 个存储单元起就是实际代码了，直到第 3 个存储单元所示的长度为止。

有问题了，CPU 如何访问到外设的 ROM 呢？

访问外设有两种方式。

（1）内存映射：通过地址总线将外设自己的内存映射到某个内存区域（并不是映射到主板上插的内存条中）。

（2）端口操作：外设都有自己的控制器，控制器上有寄存器，这些寄存器就是所谓的端口，通过 in/out 指令读写端口来访问硬件的内存。

控制显卡用的便是内存映射+端口操作的方式，这个以后会在操作显卡时介绍。

从内存的物理地址 0xA0000 开始到 0xFFFFF 这部分内存中，一部分是专门用来做映射的，如果硬件存在，硬件自己的 ROM 会被映射到这片内存中的某处，至于如何映射过去的，咱们暂时先不要深入了，这是硬件完成的工作。

如图 0-11 所示，BIOS 在运行期间会扫描 0xC0000 到 0xE0000 之间的内存，若在某个区域发现前两个字节是 0x55 和 0xAA 时，这意味着该区域对应的 rom 中有代码存在，再对该区域做累加和检查，若结果与第 3 个字节的值相符，说明代码无误，就从第 4 个字节进入。这时开始执行了硬件自带的例程以初始化硬件自身，最后，BIOS 填写中断向量表中相关项，使它们指向硬件自带的例程。

ROM Area

start	end	size	region/exception	description
Standard usage of the ROM Area				
0x000A0000	0x000BFFFF	128 KiB	video RAM	VGA display memory
0x000C0000	0x000C7FFF	32 KiB (typically)	ROM	Video BIOS
0x000C8000	0x000EFFFF	160 KiB (typically)	ROMs and unusable space	Mapped hardware & Misc.
0x000F0000	0x000FFFFF	64 KiB	ROM	Motherboard BIOS

▲图 0-11　rom area

中断向量表中第 0H～1FH 项是 BIOS 中断。

有没有新的疑问？外设的内存是如何被映射的？我也不知道，这是早期硬件工程师们大胆且天才的做法，他们在很久以前就解决了。有知道的同学希望你告诉我，哈哈，在这里，我就先当它是我的公设了。

另外，上面说的是 BIOS 在填写中断向量表，那该表是谁创建的呢？答案就是 CPU 原生支持的，不用谁负责创建。之前我曾说过，软件是靠硬件来运行的，软件能实现什么功能，很大程度上取决于硬件提供了哪些支持。软件中只要执行 int 中断向量号，CPU 便会把向量号当作下标，去中断向量表中定位中断处理程序并执行。

如果哪位同学想查看下 BIOS 在中断向量表 IVT 中建立了哪些中断例程，可以在虚拟机 bochs 或 qume 中查看，我在这里贴个表，即表 0-2，大家可以先了解下。

表 0-2　　中断向量表

中断向量	中断处理例程地址	中断描述
INT# 00	F000:FF53 (0x000fff53)	DIVIDE ERROR ; dummy iret
INT# 01	F000:FF53 (0x000fff53)	SINGLE STEP ; dummy iret
INT# 02	F000:FF53 (0x000fff53)	NON-MASKABLE INTERRUPT ; dummy iret
INT# 03	F000:FF53 (0x000fff53)	BREAKPOINT ; dummy iret
INT# 04	F000:FF53 (0x000fff53)	INT0 DETECTED OVERFLOW ; dummy iret
INT# 05	F000:FF53 (0x000fff53)	BOUND RANGE EXCEED ; dummy iret
INT# 06	F000:FF53 (0x000fff53)	INVALID OPCODE ; dummy iret
INT# 07	F000:FF53 (0x000fff53)	PROCESSOR EXTENSION NOT AVAILABLE ; dummy iret
INT# 08	F000:FEA5 (0x000ffea5)	IRQ0 - SYSTEM TIMER
INT# 09	F000:E987 (0x000fe987)	IRQ1 - KEYBOARD DATA READY
INT# 0a	F000:E9DF (0x000fe9df)	IRQ2 - LPT2
INT# 0b	F000:E9DF (0x000fe9df)	IRQ3 - COM2
INT# 0c	F000:E9DF (0x000fe9df)	IRQ4 - COM1
INT# 0d	F000:E9DF (0x000fe9df)	IRQ5 - FIXED DISK
INT# 0e	F000:EF57 (0x000fef57)	IRQ6 - DISKETTE CONTROLLER
INT# 0f	F000:E9DF (0x000fe9df)	IRQ7 - PARALLEL PRINTER
INT# 10	C000:014A (0x000c014a)	VIDEO
INT# 11	F000:F84D (0x000ff84d)	GET EQUIPMENT LIST
INT# 12	F000:F841 (0x000ff841)	GET MEMORY SIZE
INT# 13	F000:E3FE (0x000fe3fe)	DISK

续表

中断向量	中断处理例程地址	中断描述
INT# 14	F000:E739 (0x000fe739)	SERIAL
INT# 15	F000:F859 (0x000ff859)	SYSTEM
INT# 16	F000:E82E (0x000fe82e)	KEYBOARD
INT# 17	F000:EFD2 (0x000fefd2)	PRINTER
INT# 18	F000:969B (0x000f969b)	CASETTE BASIC
INT# 19	F000:E6F2 (0x000fe6f2)	BOOTSTRAP LOADER
INT# 1a	F000:FE6E (0x000ffe6e)	TIME
INT# 1b	F000:FF53 (0x000fff53)	KEYBOARD - CONTROL-BREAK HANDLER ; dummy iret
INT# 1c	F000:FF53 (0x000fff53)	TIME - SYSTEM TIMER TICK ; dummy iret
INT# 1d	0000:0000 (0x00000000)	SYSTEMDATA-VIDEO PARAMETER TABLES
INT# 1e	F000:EFDE (0x000fefde)	SYSTEM DATA - DISKETTE PARAMETERS
INT# 1f	C000:1378 (0x000c1378)	SYSTEM DATA - 8x8 GRAPHICS FONT
INT# 20	F000:FF53 (0x000fff53)	; dummy iret
INT# 21	F000:FF53 (0x000fff53)	; dummy iret
INT# 22	F000:FF53 (0x000fff53)	; dummy iret
INT# 23	F000:FF53 (0x000fff53)	; dummy iret
INT# 24	F000:FF53 (0x000fff53)	; dummy iret
INT# 25	F000:FF53 (0x000fff53)	; dummy iret
INT# 26	F000:FF53 (0x000fff53)	; dummy iret
INT# 27	F000:FF53 (0x000fff53)	; dummy iret
INT# 28	F000:FF53 (0x000fff53)	; dummy iret
INT# 29	F000:FF53 (0x000fff53)	; dummy iret
INT# 2a	F000:FF53 (0x000fff53)	; dummy iret
INT# 2b	F000:FF53 (0x000fff53)	; dummy iret
INT# 2c	F000:FF53 (0x000fff53)	; dummy iret
INT# 2d	F000:FF53 (0x000fff53)	; dummy iret
INT# 2e	F000:FF53 (0x000fff53)	; dummy iret
INT# 2f	F000:FF53 (0x000fff53)	; dummy iret
INT# 30	F000:FF53 (0x000fff53)	; dummy iret
INT# 31	F000:FF53 (0x000fff53)	; dummy iret
INT# 32	F000:FF53 (0x000fff53)	; dummy iret
INT# 33	F000:FF53 (0x000fff53)	; dummy iret
INT# 34	F000:FF53 (0x000fff53)	; dummy iret
INT# 35	F000:FF53 (0x000fff53)	; dummy iret
INT# 36	F000:FF53 (0x000fff53)	; dummy iret
INT# 37	F000:FF53 (0x000fff53)	; dummy iret
INT# 38	F000:FF53 (0x000fff53)	; dummy iret
INT# 39	F000:FF53 (0x000fff53)	; dummy iret
INT# 3a	F000:FF53 (0x000fff53)	; dummy iret
INT# 3b	F000:FF53 (0x000fff53)	; dummy iret
INT# 3c	F000:FF53 (0x000fff53)	; dummy iret
INT# 3d	F000:FF53 (0x000fff53)	; dummy iret
INT# 3e	F000:FF53 (0x000fff53)	; dummy iret
INT# 3f	F000:FF53 (0x000fff53)	; dummy iret
INT# 40	F000:EC59 (0x000fec59)	
INT# 41	9FC0:003D (0x0009fc3d)	
INT# 42	F000:FF53 (0x000fff53)	; dummy iret
INT# 43	C000:2578 (0x000c2578)	
INT# 44	F000:FF53 (0x000fff53)	; dummy iret

续表

中断向量	中断处理例程地址	中断描述
INT# 45	F000:FF53 (0x000fff53)	; dummy iret
INT# 46	9FC0:004D (0x0009fc4d)	
INT# 47	F000:FF53 (0x000fff53)	; dummy iret
INT# 48	F000:FF53 (0x000fff53)	; dummy iret
INT# 49	F000:FF53 (0x000fff53)	; dummy iret
INT# 4a	F000:FF53 (0x000fff53)	; dummy iret
INT# 4b	F000:FF53 (0x000fff53)	; dummy iret
INT# 4c	F000:FF53 (0x000fff53)	; dummy iret
INT# 4d	F000:FF53 (0x000fff53)	; dummy iret
INT# 4e	F000:FF53 (0x000fff53)	; dummy iret
INT# 4f	F000:FF53 (0x000fff53)	; dummy iret
INT# 50	F000:FF53 (0x000fff53)	; dummy iret
INT# 51	F000:FF53 (0x000fff53)	; dummy iret
INT# 52	F000:FF53 (0x000fff53)	; dummy iret
INT# 53	F000:FF53 (0x000fff53)	; dummy iret
INT# 54	F000:FF53 (0x000fff53)	; dummy iret
INT# 55	F000:FF53 (0x000fff53)	; dummy iret
INT# 56	F000:FF53 (0x000fff53)	; dummy iret
INT# 57	F000:FF53 (0x000fff53)	; dummy iret
INT# 58	F000:FF53 (0x000fff53)	; dummy iret
INT# 59	F000:FF53 (0x000fff53)	; dummy iret
INT# 5a	F000:FF53 (0x000fff53)	; dummy iret
INT# 5b	F000:FF53 (0x000fff53)	; dummy iret
INT# 5c	F000:FF53 (0x000fff53)	; dummy iret
INT# 5d	F000:FF53 (0x000fff53)	; dummy iret
INT# 5e	F000:FF53 (0x000fff53)	; dummy iret
INT# 5f	F000:FF53 (0x000fff53)	; dummy iret
INT# 60	0000:0000 (0x00000000)	此项为空，未添加中断处理例程
INT# 61	0000:0000 (0x00000000)	此项为空，未添加中断处理例程
INT# 62	0000:0000 (0x00000000)	此项为空，未添加中断处理例程
INT# 63	0000:0000 (0x00000000)	此项为空，未添加中断处理例程
INT# 64	0000:0000 (0x00000000)	此项为空，未添加中断处理例程
INT# 65	0000:0000 (0x00000000)	此项为空，未添加中断处理例程
INT# 66	0000:0000 (0x00000000)	此项为空，未添加中断处理例程
INT# 67	0000:0000 (0x00000000)	此项为空，未添加中断处理例程
INT# 68	F000:FF53 (0x000fff53)	; dummy iret
INT# 69	F000:FF53 (0x000fff53)	; dummy iret
INT# 6a	F000:FF53 (0x000fff53)	; dummy iret
INT# 6b	F000:FF53 (0x000fff53)	; dummy iret
INT# 6c	F000:FF53 (0x000fff53)	; dummy iret
INT# 6d	F000:FF53 (0x000fff53)	; dummy iret
INT# 6e	F000:FF53 (0x000fff53)	; dummy iret
INT# 6f	F000:FF53 (0x000fff53)	; dummy iret
INT# 70	F000:FE93 (0x000ffe93)	IRQ8 - CMOS REAL-TIME CLOCK
INT# 71	F000:E9D6 (0x000fe9d6)	IRQ9 - REDIRECTED TO INT 0A BY BIOS
INT# 72	F000:E9E5 (0x000fe9e5)	IRQ10 - RESERVED
INT# 73	F000:E9E5 (0x000fe9e5)	IRQ11 - RESERVED
INT# 74	F000:95C9 (0x000f95c9)	IRQ12 - POINTING DEVICE

续表

中断向量	中断处理例程地址	中断描述
INT# 75	F000:E2C7 (0x000fe2c7)	IRQ13 - MATH COPROCESSOR EXCEPTION
INT# 76	F000:9A60 (0x000f9a60)	IRQ14-HARD DISK CONTROLLER OPERATION COMPLETE
INT# 77	F000:E9E5 (0x000fe9e5)	IRQ15-SECONDARYIDE CONTROLLER OPERATION COMPLETE
INT# 78	0000:0000 (0x00000000)	此项为空，未添加中断处理例程

DOS 是运行在实模式下的，故其建立的中断调用也建立在中断向量表中，只不过其中断向量号和 BIOS 的不能冲突。

0x20～0x27 是 DOS 中断。因为 DOS 在实模式下运行，故其可以调用 BIOS 中断。

DOS 中断只占用 0x21 这个中断号，也就是 DOS 只有这一个中断例程。

DOS 中断调用中那么多功能是如何实现的？是通过先往 ah 寄存器中写好子功能号，再执行 int 0x21。这时在中断向量表中第 0x21 个表项，即物理地址 0x21*4 处中的中断处理程序开始根据寄存器 ah 中的值来调用相应的子功能。

而 Linux 内核是在进入保护模式后才建立中断例程的，不过在保护模式下，中断向量表已经不存在了，取而代之的是中断描述符表（Interrupt Descriptor Table，IDT）。该表与中断向量表的区别会在讲解中断时详细介绍。所以在 Linux 下执行的中断调用，访问的中断例程是在中断描述符表中，已不在中断向量表里了。

Linux 的系统调用和 DOS 中断调用类似，不过 Linux 是通过 int 0x80 指令进入一个中断程序后再根据 eax 寄存器的值来调用不同的子功能函数的。再补充一句：如果在实模式下执行 int 指令，会自动去访问中断向量表。如果在保护模式下执行 int 指令，则会自动访问中断描述符表。

以上主要对 BIOS 中断多介绍了一点，尽管对 DOS 说得不多，不过有了 BIOS 中断的表述，相信同学们对 DOS 中断调用也清楚了，其原理介于 BIOS 中断调用和 Linux 中断调用之间。后面在实现系统调用时，全是基于 Linux 思想的，所以在此对 Linux 系统调用的介绍点到为止。

0.21 Section 和 Segment 的区别

C 程序大体上分为预处理、编译、汇编和链接 4 个阶段。预处理阶段是预处理器将高级语言中的宏展开，去掉代码注释，为调试器添加行号等。编译阶段是将预处理后的高级语言进行词法分析、语法分析、语义分析、优化，最后生成汇编代码。汇编阶段是将汇编代码编译成目标文件，也就是转换成了目标机器平台上的机器指令。链接阶段是将目标文件连接成可执行文件。这里我们只关注汇编和链接这两个阶段。

在汇编源码中，通常用语法关键字 section 或 segment 来表示一段区域，它们是编译器提供的伪指令，作用是相同的，都是在程序中“逻辑地”规划一段区域，此区域便是节。注意，此时所说的 section 或 segment 都是汇编语法中的关键字，它们在语法中都表示“节”，不是段，只是不同编译器的关键字不同而已，关键字 segment 在语法中也被认为与 section 意义相同。首先汇编器根据语法规则，会将汇编源码中表示“节”的语法关键字 section 或 segment 在目标文件中编译成“节”，此“节”便是我们要讨论的 section。经过汇编生成目标文件之后，由这些 section 或 segment 修饰的程序区域便成为了“节”（section）。但操作系统加载程序时并不关心节的数量和大小，操作系统只关心节的属性，因为程序必然是要加载到内存中才能运行的，而内存的访问会涉及到全局描述符表中段描述符的访问权限等属性，保护模式下对任何内存的访问都要经过段描述符才行。比如程序代码所在的段描述符权限属性必须是只读，数据所在的段描述符的权限属性必然是可读写，程序中那些只读的节（比如代码区域）必然不能指向可读写的段描述符，同样，程序中的数据也不能用只读权限的段描述符去访问。如果此时您对段描述符不了解，以后咱们在介绍保护模式下全局描述表时就明白了。操作系统在加载程序时，不需要对逐个节进行加载，只要给出相同权限的节的集合就行了，例如把所有只读可执行的节（如代码节.text 和初始化代码节.init）归并到一块，所有可读写的节（如数据节.data 和未初始化节.bss）归并到

一块，这样操作系统就能为它们分配不同的段选择子，从而指向不同段描述符，实现不同的访问权限了。为了程序能在操作系统上运行，操作系统和编译器需要相互配合，此时汇编器只生成了目标文件，尚未链接，因此这个将“节”合并的工作是由链接器来完成的，链接器将目标文件中属性相同的节合并成一个大的 section 集合，此集合便称为 segment，也就是段，此段便是我们平时所说的可执行程序内存空间中的代码段和数据段。

现在总结一下。

section 称为节，是指在汇编源码中经由关键字 section 或 segment 修饰、逻辑划分的指令或数据区域，汇编器会将这两个关键字修饰的区域在目标文件中编译成节，也就是说“节”最初诞生于目标文件中。

segment 称为段，是链接器根据目标文件中属性相同的多个 section 合并后的 section 集合，这个集合称为 segment，也就是段，链接器把目标文件链接成可执行文件，因此段最终诞生于可执行文件中。我们平时所说的可执行程序内存空间中的代码段和数据段就是指的 segment。

在大多数情况下，这两者都被混为一谈，现在咱们做个实际测试，通过实验结果来展示出这两者的不同。其实用一个测试样例就能得出结果，不过为了消除大家的疑虑，测试得更彻底一点，在这里给大家准备了两个小汇编文件，将它们编译链接后，我们通过 readelf 命令查看其信息来得出结论。上菜了。

文件 1.asm

```
[work@localhost test]$ cat 1.asm
section .bss
resb    2 * 32

section file1data       ; 自定义的数据段，未使用“传统”的.data

strHello   db   "Hello, world!", 0Ah
STRLEN      equ  $ - strHello

section file1text       ; 自定义的代码段，未使用“传统”的.text
extern print            ; 声明此函数在别的文件中，
                        ; 告诉编译器在编译本文件时找不到此符号也没关系，在链接时会找到
global _start           ; 链接器把_start当作程序的入口

_start:
      push STRLEN       ; 传入参数，字符串长度
      push strHello     ; 传入参数，待打印的字符串
      call print        ; 此函数定义在2.asm

;返回到系统
   mov ebx, 0           ; 返回值4
   mov eax, 1           ; 系统调用号1:sys_exit
   int 0x80             ; 系统调用
```

这个汇编文件是在本地中声明了字符串，并调用外部的打印函数 print，大家可以参考注释，弄个大概明白就行。

文件 2.asm

```
[work@localhost test]$ cat 2.asm

section .text
mov eax , 0x10
jmp $


section file2data       ; 自定义的数据段，未使用“传统”的.data义数据段

file2vardb    3



section file2text       ; 自定义的代码段，未使用“传统”的.text

global print            ; 导出print,供其他模块使用

print:
   mov    edx,[esp+8]   ; 字符串长度
   mov    ecx, [esp+4]  ; 字符串

   mov ebx, 1
   mov eax, 4           ; sys_write
   int0x80    ;         ; 系统调用
   ret
```

在文件 2.asm 中声明了函数 print。下面将这两个文件分别编译成 elf 格式，这样方便我们通过 readelf 来查看其编译结果。开始编译，链接成可执行文件 12。

```
[work@localhost test]$nasm -f elf 1.asm -o 1.o
[work@localhost test]$nasm -f elf 2.asm -o 2.o
[work@localhost test]$ld 1.o 2.o -o 12
```

没问题，再执行一下。

```
[work@localhost test]$ ./12
Hello，world!
```

打印出了 Hello，world!，结果正确。让我们用 readelf 查看下文件 12 的头信息，如图 0-12 所示。

```
[work@localhost test]$ readelf -e ./12
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048095
  Start of program headers:          52 (bytes into file)
  Start of section headers:          276 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         10
  Section header string table index: 7

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        08048080 000080 000007 00  AX  0   0 16
  [ 2] file1data         PROGBITS        08048087 000087 00000e 00   A  0   0  1
```

readelf 输出信息 1

```
  [ 3] file1text         PROGBITS        08048095 000095 000018 00   A  0   0  1
  [ 4] file2data         PROGBITS        080480ad 0000ad 000001 00   A  0   0  1
  [ 5] file2text         PROGBITS        080480ae 0000ae 000015 00   A  0   0  1
  [ 6] .bss              NOBITS          080490c4 0000c4 000040 00  WA  0   0  4
  [ 7] .shstrtab         STRTAB          00000000 0000c3 00004e 00      0   0  1
  [ 8] .symtab           SYMTAB          00000000 0002a4 000110 10      9  12  4
  [ 9] .strtab           STRTAB          00000000 0003b4 00004b 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x000c3 0x000c3 R E 0x1000
  LOAD           0x0000c4 0x080490c4 0x080490c4 0x00000 0x00040 RW  0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .text file1data file1text file2data file2text
   01     .bss
```

readelf 输出信息 2

▲图 0-12　头信息

结果好长，为了方便查看，我对关键部分加以注释，如图 0-13 和图 0-14 所示。

在上面重点部分我都用文字标出了，要注意 section headers 的部分，此部分显示可执行文件中所有的

section，也包括我们在两个汇编文件中用关键字 section 定义的部分。从第 2 个 section 到第 5 个 section，是 1.asm 中的自定义数据 section: file1data，自定义代码 section: file1text 和 2.asm 中的自定义数据 section: file2data 和自定义代码 section: file2text。

再往下看 Program Headers 部分，此处一共有两个段，第一个段是我们的代码段，通过其 Flg 值为 RE 便可推断，只读（Readonly）可执行（Execute），其 MemSiz 为 0x000c3。此段对应 Section to Segment mapping 部分中的第 00 个 Segment，此 segment 中包括 section: .text file1data file1text file2data file2text。

```
Section Headers:
  [Nr] Name          Type        Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]               NULL        00000000 000000 000000 00      0   0  0
  [ 1] .text         PROGBITS    08048080 000080 000007 00  AX  0   0 16

  以下2、3、4、5是需要关注的部分，这是源程序中定义的section部分
  可见，elf中的section部分，便是源程序中定义的section

  [ 2] file1data     PROGBITS    08048087 000087 00000e 00   A  0   0  1
  [ 3] file1text     PROGBITS    08048095 000095 000018 00   A  0   0  1
  [ 4] file2data     PROGBITS    080480ad 0000ad 000001 00   A  0   0  1
  [ 5] file2text     PROGBITS    080480ae 0000ae 000015 00   A  0   0  1

                 下面的输出暂且忽略
                  [ 6] .bss        NOBITS     080490c4 0000c4 000040 00  WA  0   0  4
                  [ 7] .shstrtab   STRTAB     00000000 0000c3 00004e 00      0   0  1
                  [ 8] .symtab     SYMTAB     00000000 0002a4 000110 10      9  12  4
                  [ 9] .strtab     STRTAB     00000000 0003b4 00004b 00      0   0  1
                 Key to Flags:
                  W (write), A (alloc), X (execute), M (merge), S (strings)
                  I (info), L (link order), G (group), x (unknown)
                  O (extra OS processing required) o (OS specific), p (processor specific)

重点又来了，程序中有两个segment，也就是Program Headers,flg分别是 R E 和RW,可推测
第1个为只读可执行的代码段，第2个是可读写的数据段

Program Headers:
  Type      Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD      0x000000 0x08048000 0x08048000 0x000c3 0x000c3 R E 0x1000
  LOAD      0x0000c4 0x080490c4 0x080490c4 0x00000 0x00040 RW  0x1000

这里是section映射到 segment，也就是多个section归并到segment中
```

▲图 0-13　节和段

```
Section to Segment mapping:
Segment Sections...
  可见下面在section header中显示的节被归入了第1个段，也就是上面的flg为R E的只读可执行的代码段
  左边这列是segment号，右边这列是左边segment中包含的所有section
Segment      Sections...
  00     .text file1data file1text file2data file2text

  下面这个.bss就单独归为一个段
  01     .bss
```

▲图 0-14　节合并到段

第二个段便是我们的数据段，但此数据段中只包含.bss 节（section），它用于存储全局未初始化数据，故其 Flg 必然可读写，其属性为 RW。此段 MemSiz 大小为 0x40，即十进制的 64，可见，这和 1.asm 中定义的 bss 大小一致，而在 2.asm 中未定义.bbs section，所以此 bss 指的就是 1.asm 中的定义。此段对应 Section to Segment mapping 部分中的第 01 个 Segment，而此 segment 只包括.bss 节，独立成一个段了。

到此文件分析完毕，总结一下。

自定义的 section 名，会在 elf 的 section header 中显示出来。下面是几个标准的 section（节）名，不是 segment（段）名，segment 没有名称。

```
节名                  说明
.data         用于存入数据，可读可写
```

```
.text       用于存入代码，只读可执行
.bss        全局未初始化区域
```

在汇编代码中，若以标准节名定义 section，如我们定义的.bss 便是标准节名。编译器会按照以上说明中的要求使用 section 内的数据。

不管定义了多少节名，最终要把属性相同的 section，或者编译认为可以放到一块的，合并到一个大的 segment 中，也就是 elf 中说的 program header 中的项。由此可见，某个节（section）属于某个段（segment），段是由节组成的。另外多说一句，最终给加载器用的也是 program header 中显示的段，这才是进程的资源，这部分内容将在加载内核时展开。在第 3 章中介绍了 section 在地址分配上的内容，大家有兴趣可以提前了解下。

0.22 什么是魔数

魔数，magic number，这让一部分人感觉到迷惑，也让另一部分人迷惑。哈哈，两个迷惑，把我们都搞迷惑了，作者你到底想表达什么意思啊。没错，其实魔数的本意就是让人感到迷惑的数，看到某个数，不知道其代表何意，用东北话说，都蒙圈了。一部分人对这个概念迷惑的原因是这有什么好解释的，一种司空见惯的东西，即使不知道是怎么来的，但由于大脑经常被其训练，对其已经形成深刻的印象，似乎理所当然地接受了。当我向别人请教一个类似的问题时，如果被回复“这是规定”时，我就很无语。任何规定都是出自于某种原因才做出的，很少有规定是靠拍脑门或抓阄决定的。就像国外的电视剧，一部称为一季，季是由 season 翻译过来的，表示季节，一个时段。一个季节过去了，这和电视剧整体情节暂告一段落是一样的，这较容易理解。

另一部分人感到迷惑的原因是真心想搞清楚概念是什么意思，我也属于这一类。

魔数，其实也称为神奇数字，我们大多数人是在学习计算机过程中接触到这个词的。它被用来为重要的数据定义标签，用独特的数字唯一地标识该数据，这种独特的数字是只有少数人才能掌握其奥秘的“神秘力量。”

对魔数简单的阐述就是：不明就理地出现一个数字，不知道其是什么意思，感觉看不透，猜不出，就像魔法一样很神秘。了解一定上下文的人肯定知道是什么意思，一般局外人绞尽脑汁也不解其意。就像小姑娘对着小伙子伸出大拇指和食指，小伙子马上就意会了，这是让我晚上 8 点在村口东边老槐树下见。

如果程序中出现这样的代码：

```
int a = 2014 – 1987;
```

根据直觉，似乎这是在求年龄，因为 2014 是和现在很接近的年份，而 1987 似乎是生日。但这只是主观估计，万一这两个数字表示的是这个月和上个月的电表计数呢，人家在查电费不行吗……修改一下代码。

```
#define birthday 1987;
int a = 2014 – birthday;
```

由于 1987 用了一个宏代替，即使变量名称不改为 age，还叫作 a，大家也明确了这是在求年纪呢。

故，直接出现的一个数字，只要其意义不明确，感觉很诡异，就称之为魔数。魔数应用的地方太多了，如 elf 文件头。

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
```

这个 Magic 后面的一长串就是魔数，elf 解析器（通常是程序加载器）用它来校验文件的类型是否是 elf。

主引导记录最后的两个字节的内容是 0x55，0xaa，这表明这个扇区里面有可加载的程序，BIOS 就用它来校验该扇区是否可引导。

有人说只要为这些数字赋予实际的意义不就行了吗。其实，无论怎么给这组陌生的数字赋予名称，它都不像熟悉的出生日期那样直观易懂（如对于 19590318，不解释大家也会知道 0318 是 3 月 18 日），反而还要额外增加一些内容来解释，得不偿失，所以这就是魔数不得不存在的原因。

可见，计算机中处处是协议、约定。不过为了程序意义清晰可维护性强，尽量还是少用魔数。

0.23 操作系统是如何识别文件系统的

我们知道，一个硬盘上可以有很多分区，每个分区的格式又可以不同。就拿 Linux 来说，既能识别 ext3，又能识别 ext4。可能有同学会说，这两个分区的文件系统都是 Linux 自己专用的，当然认得自己的东西了。可是自己的东西也得有个辨别的地方，否则凭什么说“认得”呢。

其实这是之前介绍过的魔数的作用，文件系统也有自己的魔数，魔数的神秘力量在此施展了。各分区都有超级块，一般位于本分区的第 2 个扇区，比如若各分区的扇区以 0 开始索引，其第 1 个扇区便是超级块的起始扇区。超级块里面记录了此分区的信息，其中就有文件系统的魔数，一种文件系统对应一个魔数，比对此值便知道文件系统类型了。

0.24 如何控制 CPU 的下一条指令

其实此问题我一直犹豫要不要写出来，因为大部人都觉得这个问题有些匪夷所思，CPU 是负责执行指令的，它会按照程序的执行流程走，此问题的目的其实就是想知道如何牵着 CPU 的鼻子走。当初我被问这个问题时也觉得很诧异，甚至我觉得自己可能没理解人家的意思。后来他这样跟我说：“CPU 要执行的下一条指令是在 CS：IP 寄存器吧?”我说：“是啊”。他又问：“CS 和 IP 寄存器，是用 mov 指令修改的吗？”我听后，顿时觉得他这个问题很有意义，暗自对他有些小敬佩，我相信很多人都没想过，CS 和 IP 能不能用 mov 指令去修改。

是这样的，我们常说的用于存放下一条指令地址的寄存器称为程序计数器 PC（Program Counter）。这个名词在我看来是个概念级别的内容，它只是 CPU 中有关下一条指令存放地址的统称，也就是说 PC 是用来表示下一条指令的存放地址，具体的实现形式不限，后面会有所讨论。

CPU 按照指令集可以分为很多种，由于 PC 只是个概念，所以在不同种类的 CPU 中，有不同的实现。注意啦，这里的“不同种类”不是指 CPU 品牌，而是指 CPU 体系结构，如 INTEL 和 AMD 同属 x86 构架，如果您对此不了解，细心的我早已在下面为您准备好了体系结构、指令集的相关内容。由于此方面内容较独立，我专门将其组织成一个小节供大伙儿参考，如果您现在感兴趣，可以先参阅“指令集、体系结构、微架构、编程语言”这一节。

在 x86 体系结构的 CPU 中，也就是咱们大多数人使用的 INTEL 或 AMD 公司出品的桌面处理器，程序计数器 PC 并不是单一的某种寄存器，它是一种寄存器组合，指的段寄存器 CS 和指令指令寄存器 IP。

CS 和 IP 是 CPU 待执行的下一条指令的段基址和段内偏移地址，不能直接用 mov 指令去改变它们，我想可能的一个原因是：mov 指令一次只能改变一个寄存器，不能同时将 cs 和 ip 都改变。如果只改变了其中一个会引起错误。如改变了 cs 的值后，ip 的值还是原先 cs 段的偏移，很难保证新的 cs 段内的偏移地址 ip 处的指令是正确的。因此，有专门改变执行流的指令，如 jmp、call、int、ret，这些指令可以同时修改 cs 和 ip，它们在硬件级别上实现了原子操作。

以上说的是 x86 体系的 CPU，其他类型的 CPU 是怎样的呢？这就取决于具体实现啦，咱们这里拿 ARM 举例，它的程序计数器有个专门的寄存器，名字就叫 PC，想要改变程序流程，直接对该寄存器赋值便可。

与 x86 不同的是在 ARM 中可以用 mov 指令来修改程序流，在 ARM 体系 CPU 的汇编器中，寄存器的名称在汇编语言中是以“r 数字”的形式命名的，例如汇编代码：mov pc，r0，表示将寄存器 r0 中的内容赋值给程序寄存器 PC，这样就直接改变了程序的执行流。

总结一下，程序计数器 PC 负责处理器的执行方向，它只是获取下一条指令的方法形式，在不同体系结构的 CPU 中有不同的实现方法。

0.25 指令集、体系结构、微架构、编程语言

指令集是什么？表面上看它是一套指令的集合。集合的意思显而易见，那咱们说说什么是指令。

在计算机中，CPU 只能识别 0、1 这两个数，甚至它都不知道数是什么，它只知道要么“是”，要么“不是”，恰好用 0、1 来表示这两种状态而已。

人发明的东西逃不出人的思维，所以，先看看我们人类的语言是怎么回事。

不同的语言对同一种事物有不同的名字，这个名字其实就是代码。比如说人类的好朋友：狗，咱们在中文里称之为狗，但在英文中它被称为 dog，虽然用了两种语言，但其描述的都是这种会汪汪叫、对人类无比忠诚的动物。人是怎样识别小狗的呢？识别信息来自听觉、视觉等，这是因为人天生具备处理声音和图像的能力，能够识别出各种不同的声音和颜色不同的图像。可是计算机只能处理 0、1 这两个数，所以让计算机识别某个事物，只有用 01 这两个数来定义。也就是说，要用 0、1 来为各种事物编码。

为了更好地说明指令集，咱们这里不再用现有的语言举例子，当然也不是要自创指令集。下面举个简单的例子来演示指令集的模型。

咱们拿表达式 A=B+C 为例。假设 A、B、C 都是内存变量的值，它们的地址分别是 0x3000、0x3004、0x3008。在此用 Ra 表示寄存器 A，Rb 表示寄存器 B，Rc 表示寄存器 C。

完成这个加法的步骤是先将 B 和 C 载入到 Ra 和 Rb 寄存器中，再将两个寄存器的值相加后送入寄存器 Ra，之后再将寄存器 Ra 的值写入到地址为 0x3000 的内存中。

步骤有了，咱们再设计完成这些步骤的指令。

步骤 1：将内存中的数据载入到寄存器，咱们假设它的指令名为 load。

操作码	寄存器操作数 1	寄存器操作数 2	寄存器操作数 3	立即数

步骤 2：两个寄存器的加法指令，假设指令名为 add。

步骤 3：将寄存器中的内容存储到内存，假设指令名为 store。

以上指令名都是假设的，名字可以任意取，因为 CPU 不识别指令名。指令名是编译器用来给人看的，为的是方便人来编程，CPU 它只认编码。目前 CPU 中的指令，无论是哪种指令集，都由操作码和操作数两部分组成（有些指令即使指令格式中没有列出操作数，也会有隐含的操作数）。咱们也采用这种操作码+操作数的思路，分别为这两部分编码。

咱们先为操作码设计编码。

操作码名称	二进制编码
load	00
add	01
store	10

接下来为操作数编码，操作数一般是立即数、寄存器、内存等，咱们这里主要是为寄存器编码。

寄存器名称	二进制编码
Ra	00
Rb	01
Rc	10

好啦，操作码和操作数都有了，其实指令集已经完成了。不过在一长串的二进制 01 中，哪些是操作码，哪些是操作数呢？这就是指令格式的由来啦。我们人为规定个格式，规定操作码和操作数的大小及位置，然后在 CPU 硬件电路中写死这些规则，让 CPU 在硬件一级上识别这些格式，从而能识别出操作码和操作数。

假设我们的指令格式最大支持三个寄存器参数和一个立即数参数。其中操作码和各寄存器操作数各占 1 字节，立即数部分占 4 字节。各条指令并不是完全按照此格式填充，不同的指令有不同的参数，只有操作码部分是固定的，其他操作数部分是可选的。当 CPU 在译码阶段识别出操作码后，CPU 自然知道该指令需要什么样的操作数，这是写死在硬件电路中的，所以不同的指令其机器码长度很可能不一致。

为了演示指令集模型，我们在上面假设了寄存器名、指令名、格式。按理说这对于指令集来说已经全了，不过，为方便咱们了解编译器，不如咱们再假设个指令的语法吧，咱们这里学习 Intel 的语法格式：“指

令目的操作数，源操作数”。目的操作数在左，源操作数在右，此赋值顺序比较直观。Intel 想表达的是 a=b 这种语序，如 a=b，便是 mov a，b。

以上三个步骤的机器码按照十六进制表示为：

步　骤	自定义的指令	十六进制机器码
1	load Rb,0x3004	000104300000
	load Rc,0x3008	001008300000
2	add Ra,Rb,Rc	01000110
3	store 0x300c,Ra	10000c300000

以上自定义的指令便是按照咱们假设的语法来生成的。对于机器码的大小，由于指令不同，需要的操作数也不同，所以机器码大小也不同。另外，机器码中的立即数是按照 x86 架构的小端字节序写的，这一点大家要注意。小端字节序是数值中的低位在低地址，高位在高地址，数位以字节为单位。前面有一小节说明大小端字节序问题。

步骤 2 的机器码为 01 00 01 10。操作码占 1 字节，CPU 识别出第 1 字节的二进制 01 是 add 指令，知道此指令的操作数是 3 个寄存器，并且第 1 个寄存器操作数是目的寄存器，另外两个寄存器是源操作数（这都是我们假定的，并且是写死在硬件中的规则，不同的指令有不同的规则，您也可以创造出内存和寄存器混合作为操作数的加法指令）。于是到第 2 字节去读取寄存器编码，发现其值为二进制 00，就是寄存器 Ra 对应的编码。接着到下一个字节处继续读出寄存器编码，发现是二进制 01，也就是寄存器 Rb，Rc 同理。于是将寄存器 Rb 和 Rc 的值相加后存入到寄存器 Ra。

步骤 3 中，机器码为 10 00 0c300000，CPU 读取机器码的第 1 字节发现其为二进制 10，知道其为指令 store，于是便确定了，目的操作数是个立即数形式的内存地址，源操作数是个寄存器。接着到指令格式中的寄存器操作数 1 的位置去读取寄存器编码，发现其值为 00，这就是寄存器 Ra 的编码。机器码中剩下的部分便作为立即数，这样便将寄存器 Ra 的值写入到内存 0x0000300c 中了。

以上指令集的模型，确实太过于简单了，也许称之为模型都非常勉强。现实中的指令格式要远远复杂得多。下面我们看看目前世面上的指令集有哪些。

最早的指令集是 CISC（Complex Instruction Set Computer），意为复杂指令集计算机。从名字上看，这套指令集相当复杂，当初这套指令集问世的时候，它的研发者们都没想过要给它起名，只是因为后来出现了相对精简高效的指令集，所以人们为了加以区分，才将最初的这套相对复杂的指令集命名为 CISC，而后来精简高效的指令集称为 RISC（Reduced Instruction Set Computer）。

CISC 和 RISC 并不是具体的指令集，而是两种不同的指令体系，相当于指令集中的门派，是指令的设计思想。举个例子，就像中医与西医，中医讲究从整体上调理身体，西医则更多的是偏向局部。这就是两种不同的医疗思路，类似于 CISC 和 RISC 这两种指令体系。那什么是指令集呢？拿中医举例，像华佗、张仲景这两位医圣，他们虽然都是基于中医的思想治病，但医术各有特色，水平也不尽相同，这就相当于不同的指令集。一会儿咱们会介绍具体的指令集。

为什么说 CISC 复杂呢？

首先，因为它是最早的指令集，当初都是摸着石头过河，肯定有一些瑕疵在里面。其次，当初的程序员都是用汇编语言开发程序，他们当然希望汇编语言强大啦，尽量多一些指令，尽量一个指令能多干几件事，所以指令集中的指令越来越多，越来越复杂。不过这样的好处是程序员同学很爽。最后，CISC 是 Intel 使用的指令集，Intel 公司在兼容性方面做得最好，指令集在发展的过程中，还要兼容过去有瑕疵的古董，以至于最后的指令集变得有点“奇形怪状”了。

作为后起之秀的 RISC，借鉴了前辈 CISC 的经验，取其精华，弃其糟粕，当然要更好更轻量啦。它是怎么来的呢？

CISC 不是做得很全很强吗，可是很多时候，程序员并不会用到那些复杂的指令和寻址方式，即使用到了，编译器有时候为了优化，未必“全”将其编译为复杂的形式。这就导致了 CPU 中的复杂的指令和

寻址方式无用武之地。根据二八定律，指令集中 20%的简单指令占了程序的 80%，而指令集中 80%的复杂指令占了程序的 20%。根据这个特性，处理器及指令集被重新设计，保留了那些基本常用的指令，减少了硬件电路的复杂性。这样，大部分指令都能在一个时钟周期内完成，更有利于提升流水线的效率。而且，指令采用了定长编码，这样译码工作更容易了。由于其太优秀了，后来的处理器，如 MIPS，ARM，Power 都采用 RISC 指令体系，做得最好的就是 MIPS 处理器，它严格遵守 RISC 思想，业界公认其优雅。

我们常用的 CPU 是 Intel 和 AMD 公司的产品，它们用的指令集便是基于 CISC 思想的 x86。AMD 的 x86 指令架构是 Intel 授权给他们的，为区别于此，Intel 在官方手册上称自己的指令集为 IA32。

虽然 AMD 采用的也是 x86 指令集，但 Intel 可没把硬件实现方法也告诉 AMD，否则 AMD 的 CPU 和 Intel 的 CPU 不就完全一样了吗，人家 Intel 也不肯呢。指令集是一套约定，里面规定的是有哪些指令、指令的二进制编码、指令格式等，如何实现这套约定，这是硬件自己的事。打个比方，这就像和朋友约好了在某餐厅吃饭，咱是坐车去，还是走着去，这是咱们的事，与吃饭是无关的。说白了，在 Intel 的 CPU 上运行的软件也能够在 AMD 的 CPU 上运行，原因就是它们共用了同用一套指令集，也就是对二进制编码达成了共识。它们面对相同的需求，可能采取了不同的行动，但都完成了任务。比如机器码是 b80000，Intel 的 CPU 经过译码，知道这是将 0 赋值给寄存器 ax，相当于汇编语言 mov ax，0。AMD 的 CPU 在译码时，也得将此机器码认为是将 0 赋值给寄存器 ax。至于它们在物理上是怎么将 0 传入寄存器 ax 中的，这是它们各自实现的方式，与指令集无关。它们各自实现的方式，就叫微架构。

总结一下，指令集是具体的一套指令编码，微架构是指令集的物理实现方式。

发展到后来，x86 指令集越来越复杂。它本属于 CISC 体系，但由于效率低下，最终在其内部实现上采取了 RISC 内核，即一条 CISC 指令在译码时，分解成多条 RISC 指令，这样其执行效率便可与 RISC 媲美啦。

目前市面上常见的指令集有五种，除 x86 是 CISC 指令体系外，ARM、MIPS、Power、C6000 都是 RISC 指令体系的指令集。

CPU 与指令集是对应的，一种 CPU 只能识别一种指令集，所以很多 CPU 都以其支持的指令集来称呼。比如 ARM、MIPS，它们本身是 CPU 名称，又是指令集名称。

ARM 主要用在手机中，作为手机的处理器。Power 是 IBM 用于服务器上的处理器。C6000 是数字信号处理器，广泛用于视频处理。而 MIPS 虽然本身很优秀，但其在各领域起步都较晚，并没有广泛应用的领域。

由于 MIPS 本身的优越性，龙芯用的就是 mips 指令集，有没有人问，为什么咱们自主研发的 CPU 还要用人家国外的指令集？就不能也研发出一套指令集吗？能倒是能，不过语言不通用。就像我自己可以发明一门语言，语言本身没什么问题，问题是我用自己发明的语言和别人交流，谁听得懂呢，谁又愿意去学这门语言呢？大家都很忙，不通用的东西没人愿意花精力去学。如果龙芯也自立门户创造新的指令集，那有谁愿意给它写编译器呢？即使有了编译器，操作系统也要重新编译发布，应用程序也要重新编译发布，指令集背后不仅是个计算机生态链，更重要的是全球经济链。

平时所说的编程语言，虽然其上层表现各异，归根结底是要在具体的 CPU 上运行的，所以必须由编译器按照该 CPU 的指令集，翻译成符合该 CPU 的指令。说到这，不得不说一下交叉编译，本质上交叉编译就是用在 A 平台上运行的编译器，编译出符合 B 平台 CPU 指令集的程序，编译出的程序直接能在 B 平台上运行啦。这里的平台指的就是 CPU 指令体系结构。

0.26 库函数是用户进程与内核的桥梁

在讨论此问题之前，我们应该明白此问题的始作俑者是操作系统本身。我们用了操作系统，就理应遵守它的规范。任何操作系统都有自己的一套做事规则，在其上的所有应用程序，都按照它定下的规矩做事。

我们讨论的环境是 Linux，所以，以下所有的内容都是在 Linux 系统的规则之中讨论，我们所讨论的内容便是搞清楚这些规则。

在 Linux 下 C 编程时，我们写的程序通常是用户级程序。为了输出文本，我们一般会在文件开始 include <stdio.h>，这样程序就可以使用 printf 这样的函数完成打印输出。这背后的原理是什么？为什么简单包含

stdio.h 后就能够打印字符呢？

揭晓这些答案必须要交待一个事实，用户程序不具备独立打印字符的功能，它必须借助操作系统的力量才可以，如何借助呢？操作系统提供了一套系统调用接口，用户进程直接调用这些接口就行啦。简单来说，接口就是某个功能模块的入口，通过接口给该模块一个输入，它就返回一个输出，模块内部实现的过程就像个黑盒子一样，咱们看不到，也无需关心。我们能够打印字符的原因就是调用了系统调用，但是大家确实没有亲手写下调用系统调用的代码（后面章节会说），这就是库函数的功劳，它帮你写下了这些。

但我们并没有看到库函数的实现，我们只是包含了所需要的库函数所在的头文件，该头文件中有这样一句函数的声明。比如 printf 函数所在的头文件是 stdio.h，该文件位于磁盘/usr/include/目录下，其中第 361 行是对 printf 的声明。

```
extern int printf (__const char *__restrict __format,...);
```

注意上面括号中的“…”不是我人为加上的省略号，并不是函数声明太长我省略了，这是变长参数的语法。有了这句声明，咱们可以直接把它贴在调用 printf 的文件中就可以啦，不用把整个 stdio.h 包含进来了，毕竟里面声明的函数太多了，stdio.h 文件共 942 行，无关的内容太多会给我们带来困扰。

头文件被包含进来后，其内容也是原样被展开到 include 所在的位置，就是把整个头文件中的内容挪了过来，所以在头文件中的内容是什么都可以，未必一定要是函数声明，你愿意的话完全可以把函数定义在头文件中，而且也可以不用.h 作为文件名。来，咱们做个实验。

func_inc.d

```
1 void myfunc(char* str){
2    printf(str);
3 }
```

您看，我们的测试文件名为 func_inc.d，它甚至都不是以.c 结尾的。说明 include 指令不关心所包含的文件名是啥，只是原方不动地将所包含的文件内容在此处展开。它只包含这三行代码。再看函数 main.c。

main.c

```
1 extern int printf (__const char *__restrict __format, ...);
2 #include "func_inc.d"
3
4 void main() {
5   myfunc("hello world\n");
6 }
```

main.c 中第 1 行声明了外部函数 printf，平时我们 include <stdio.h>就是这个目的，只不过咱们这里让其精简了。

第 2 行将 func_inc.d 包含进来，之后第 4～6 行调用定义在 func_inc.d 中的 myfunc 函数进行打印。

不说别的，先看执行结果，如图 0-15 所示。

```
[work@localhost tmp]$ cat -n func_inc.d
     1  void myfunc(char* str){
     2     printf(str);
     3  }
[work@localhost tmp]$ cat -n main.c
     1  extern int printf (__const char *__restrict __format, ...);
     2  #include "func_inc.d"
     3
     4  void main() {
     5    myfunc("hello world\n");
     6  }
[work@localhost tmp]$ gcc -o main.bin main.c
[work@localhost tmp]$ ./main.bin
hello world
[work@localhost tmp]$
```

▲图 0-15　包含其他文件运行结果

为了证明 include 指令确实与所包含的文件名无关，咱们看看预处理后的文件内容。gcc 编译时加-E 参数就可以获取预处理后的文件内容。

```
[work@localhost tmp]$ gcc  -E main.c
# 1 "main.c"
# 1 "<built-in>"
# 1 "<命令行>"
# 1 "main.c"
extern int printf (__const char *__restrict __format, ...);
# 1 "func_inc.d" 1
void myfunc(char* str){
   printf(str);
}
# 3 "main.c" 2

void main() {
```

```
    myfunc("hello world\n");
}
[work@localhost tmp]$
```

您看到了，确实 include 功能只是将文件搬运过来。另外说明一下，如果 main.c 中添加了 include<stdio.h>，此处通过-E 生成的文件可老长了，所以咱们只加了 printf 函数的声明。

到现在为止，似乎还没有进入正题，只是想告诉大家头文件中可以写任何内容，甚至是函数体。

一下子就进入正题了，再交待另外一个事实，函数一定要有函数体才能被调用，必须有相应的函数实现，仅仅凭个头文件中的声明肯定是不行的。

如果在头文件中定义的是 printf 函数的实现，也许就容易理解头文件帮我们做了什么，可是事实不是这样的，头文件中一般仅仅有函数声明，这个声明告诉编译器至少两件事。

（1）函数返回值类型、参数类型及个数，用来确定分配的栈空间。

（2）该函数是外部函数，定义在其他文件，现在无法为其分配地址，需要在链接阶段将该函数体所在的目标文件一同链接时再安排地址。

这第二件事是我们所说的重点。

如果预处理后，主调函数所在的文件中找不到所调用函数的函数体，一定要在链接阶段把该函数体所在的目标文件链接进来，否则程序在道理上都讲不通，怎么能通过编译呢。

您看到了，main.c 中我把 func_inc.d 包含进来，include 后面并不是尖括号而是双引号“？”，这用的是自定义文件的包含，并不是包含标准文件（也就是平时我们所说的标准库头文件）。如果用了尖括号，系统就会到默认路径下去搜索该头文件。搜索到头文件后，找到其中被调函数的声明，再到另一默认文件中找该函数体的实现。

另一默认文件，按理来说应该是目标文件。它到底在哪里呢？

gcc 编译时加-v 参数会将编译、链接两个过程详细地打印出来，如图 0-16 所示。

```
 /usr/libexec/gcc/i686-redhat-linux/4.4.6/cc1 -quiet -v main.c -quiet -dumpbase main.c -mtune=
generic -march=i686 -auxbase main -version -o /tmp/ccymR62K.s
忽略不存在的目录“/usr/lib/gcc/i686-redhat-linux/4.4.6/include-fixed”
忽略不存在的目录“/usr/lib/gcc/i686-redhat-linux/4.4.6/../../../../i686-redhat-linux/include”
#include "..." 搜索从这里开始：
#include <...> 搜索从这里开始：
 /usr/local/include
 /usr/lib/gcc/i686-redhat-linux/4.4.6/include
 /usr/include
搜索列表结束。
GNU C (GCC) 版本 4.4.6 20120305 (Red Hat 4.4.6-4) (i686-redhat-linux)
        由 GNU C 版本 4.4.6 20120305 (Red Hat 4.4.6-4) 编译，GMP 版本 4.3.1，MPFR 版本 2.4.1.
GGC 准则：--param ggc-min-expand=72 --param ggc-min-heapsize=79701
Compiler executable checksum: c1037dbb624d1c27b2bd8b15ebffbe8b
COLLECT_GCC_OPTIONS='-v' '-o' 'main.bin' '-mtune=generic' '-march=i686'
 as -V -Qy -o /tmp/cc0yJGmy.o /tmp/ccymR62K.s
GNU assembler version 2.20.51.0.2 (i686-redhat-linux) using BFD version version 2.20.51.0.2-5.
34.el6 20100205
COMPILER_PATH=/usr/libexec/gcc/i686-redhat-linux/4.4.6/:/usr/libexec/gcc/i686-redhat-linux/4.4
.6/:/usr/libexec/gcc/i686-redhat-linux/:/usr/lib/gcc/i686-redhat-linux/4.4.6/:/usr/lib/gcc/i68
6-redhat-linux/:/usr/libexec/gcc/i686-redhat-linux/4.4.6/:/usr/libexec/gcc/i686-redhat-linux/:
/usr/lib/gcc/i686-redhat-linux/4.4.6/:/usr/lib/gcc/i686-redhat-linux/
LIBRARY_PATH=/usr/lib/gcc/i686-redhat-linux/4.4.6/:/usr/lib/gcc/i686-redhat-linux/4.4.6/:/usr/
lib/gcc/i686-redhat-linux/4.4.6/../../../:/lib/:/usr/lib/
COLLECT_GCC_OPTIONS='-v' '-o' 'main.bin' '-mtune=generic' '-march=i686'
 /usr/libexec/gcc/i686-redhat-linux/4.4.6/collect2 --eh-frame-hdr --build-id -m elf_i386 --has
h-style=gnu -dynamic-linker /lib/ld-linux.so.2 -o main.bin /usr/lib/gcc/i686-redhat-linux/4.4.
6/../../../crt1.o /usr/lib/gcc/i686-redhat-linux/4.4.6/../../../crti.o /usr/lib/gcc/i686-redha
t-linux/4.4.6/crtbegin.o -L/usr/lib/gcc/i686-redhat-linux/4.4.6 -L/usr/lib/gcc/i686-redhat-lin
ux/4.4.6 -L/usr/lib/gcc/i686-redhat-linux/4.4.6/../../.. /tmp/cc0yJGmy.o -lgcc --as-needed -lg
cc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed /usr/lib/gcc/i686-redhat-linu
x/4.4.6/crtend.o /usr/lib/gcc/i686-redhat-linux/4.4.6/../../../crtn.o
[work@localhost tmp]$ ./main.bin
hello world
[work@localhost tmp]$
```


▲图 0-16　gcc 编译、链接过程

gcc 内部也要将 C 代码经过编译、汇编、链接三个阶段。

（1）编译阶段是将 C 代码翻译成汇编代码，由最上面的框框中的 C 语言编译器 cc1 来完成，它将 C 代码文件 main.c 翻译成汇编文件 ccymR62K.s。

（2）汇编阶段是将汇编代码编译成目标文件，用第二个框框中的汇编语言编译器 as 完成，as 将汇编文件 ccymR62K.s 编译成目标文件 cc0yJGmy.o。

（3）链接阶段是将所有使用的目标文件链接成可执行文件，这是用左边最下面框框中的链接器 collect2 来完成的，它只是链接命令 ld 的封装，最终还是由 ld 来完成，在这一堆.o 文件中，有咱们上面的目标文件 cc0yJGmy.o。

以上我们想展开说的是第 3 点：链接阶段。

大家看到了，实际参与链接的有多个.o 文件，这些都是目标文件，也就是函数体所在的文件。printf 的函数体就在这里面其中某个.o 文件中，而且，printf 中也要调用其他函数，这些被调用的函数也分布在这些.o 文件之中。

这些咱们不认识的.o 文件从哪来？为什么链接器要链接它们？

大家看中间框框中的 LIBRARY_PATH，这是个库路径变量，里面存储的是库文件所在的所有路径，这就是编译器所说的标准库的位置，自动到该变量所包含的路径中去找库文件。以上所说的.o 文件就是在这些路径下找到的。

不知道大家注意到了没有，在图-16 中的链接阶段，链接器 collect2 的参数除了有咱们的 main.c 生成的目标文件 cc0yJGmy.o 以外，还有以下这几个以 crt 开头的目标文件：crt1.o，crti.o，crtbegin.o，crtend.o，crtn.o。

crt 是什么？CRT，即 C Run-Time library，是 C 运行时库。

什么是运行时库？

运行时库是程序在运行时所需要的库，该库是由众多可复用的函数文件组成的，由编译器提供。

所以，C 运行时库，就是 C 程序运行时所需要的库文件，在我们的环境中，它由 gcc 提供。

大家这下应该明白了，我们在程序中简单地一句 include <标准头文件>之所以有效，是因为编译器提供的 C 运行库中已经为我们准备好了这些标准函数的函数体所在的目标文件，在链接时默默帮我们链接上了。

顺便说一句，这些目标文件都是待重定位文件，重定位文件意思是文件中的函数是没有地址的，用 file 命令查看它们时会显示 relocatable，它们中的地址是在与用户程序的目标文件链接成一个可执行文件时由链接器统一分配的。所以 C 运行时库中同样的函数与不同的用户程序链接时，其生成的可执行文件中分配给库函数的地址都可能是不同的。每一个用户程序都需要与它们链接合并成一个可执行文件，所以每一个可执行文件中都有这些库文件的副本，这些库文件相当于被复制到每个用户程序中。所以您清楚了，即使咱们的代码只有十几个字符，最终生成的文件也要几 KB，就是这个道理。

还有一点内容要解释，前面说过用户程序要使用系统调用才能使用操作系统的功能，我们的 func_inc.d 中，也用到了 printf 函数，照我这么说的话，打印字符是内核的功能，那么生成的 main.bin 文件在执行 printf 函数时，内部一定会执行系统调用？没错！我们来验证一下。

我们可以用 ltrace 命令跟踪一下程序 main.bin 的执行过程就好啦。ltrace 命令用来跟踪程序运行时调用的库函数，我们的 printf 函数绝对是个标准的库函数，让我们先尝尝鲜，看看不加参数执行时的输出是否是我们想要的。走起，如图 0-17 所示。

```
[work@localhost ltrace-0.7.3]$ ~/ltrace/bin/ltrace /tmp/main.bin
__libc_start_main(0x80483d7, 1, 0xbfa6e374, 0x8048400 <unfinished ...>
printf("hello world\n"hello world
)                                                          = 12
+++ exited (status 12) +++
```

▲图 0-17 用 ltrace 跟踪进程调用的库函数

图 0-17 中用方框框出来的 printf 就是咱们调用的函数。大家机器上若没有这个命令，可以在 http://www.ltrace.org/下载，目前最新版本是 0.7.3，下载后的包是 ltrace_0.7.3.orig.tar.bz2，我把它放在了 ltrace 目录中，大家可以执行这样的命令一次性搞定。

tar jxvf ltrace_0.7.3.orig.tar.bz2 && cd ltrace-0.7.3 && ./configure --prefix=/your_path/ltrace && make && make install

验证通过之后，咱们再看看 printf 用了哪些系统调用。-S 参数查看系统调用，命令执行走起，如图 0-18 所示。

大家看到了方框中的 SYS_write 了吧，这个就是系统调用啦。Linux 的系统调用号定义在/usr/include/asm/unistd_32.h 中，大家可以自行查看。

```
[work@localhost ltrace-0.7.3]$ ~/ltrace/bin/ltrace -S /tmp/main.bin
SYS_brk(0xbfde7c6c)                                          = 0x94b8000
。。。
中间略
。。。
SYS_write(12, "hello world\n", 6794051hello world
)                                          = 12
<... printf resumed> )                                       = 12
```

▲图 0-18　用 ltrace 跟踪进程系统调用

如果大家不想安装 ltrace 命令，可以用本机自带的 strace 命令代替，它是专门用来查看系统调用和信号的命令，不过它查看的并不是最终的系统调用，而是系统调用的封装函数。不解释啦，大家眼见为实吧，如图 0-19 所示。

```
[work@localhost ~]$ strace /tmp/main.bin
execve("/tmp/main.bin", ["/tmp/main.bin"], [/* 23 vars */]) = 0
brk(0)                                 = 0x9692000
。。。
中间略
。。。
write(1, "hello world\n", 12hello world
)          = 12
exit_group(12)                         = ?
```

▲图 0-19　strace 实例

如图 0-19 所示，画框框的 write 是系统调用。原本输出的信息非常多，这里我只截了部分。write 函数是系统调用 SYS_write 的封装，所以你懂了我更喜欢用 ltrace 的原因。

顺便说一句，大家可以用-e trace=write 来限制只看 write 系统调用，免得输出无关的信息太多。

该说的都说啦，现在总结一下。

（1）操作系统有自己支持、加载用户进程的规则，而 C 运行时库是针对此操作系统的规则，为了让用户程序开发更加容易，用来支持用户进程的代码库。大家要明白，之所以我们写个程序又链接这又链接那的，完全是因为操作系统规定这样做，人在屋檐下，不得不低头。

（2）用户进程要与 C 运行时库的诸多目标文件链接后合并成一个可执行文件，也就是说我们的用户进程被加进了大量的运行库中的代码。

（3）C 运行时库作用如其名，是提供程序运行时所需要的库文件，而且还做了程序运行前的初始化工作，所以即使不包含标准库文件，链接阶段也要用到 c 运行时库。

（4）用户程序可以不和操作系统打交道，但如果需要操作系统的支持，必须要通过系统调用，它是用户进程和操作系统之间的“钩子”，用户进程顶多算是个半成品，只有通过钩子挂上了操作系统，加了上所需要的操作系统的那部分代码，用户程序才能做完一件事，这才算完整，后面章节会有详解。

（5）尽管系统调用封装在库函数中，但用户程序可以直接调用“系统调用”，不过用库函数会比较高效（后面章节会有详解）。

0.27 转义字符与 ASCII 码

计算机世界中是以二进制来运行的，无论是指令、数据，都是以二进制的形式提交给硬件处理的，字符也一样，必须转换成二进制才能被计算机识别。所以各种各样的字符编码产生，简单来说，字符编码就是用唯一的一个二进制串表示唯一的一个字符。其中最著名的字符编码就是 ASCII 码。

ASCII 码表中字符按可见分成两大类，一类是不可见字符，共 33 个，它们的 ASCII 码值是 0～31 和 127，属于控制字符或通信专用字符。表中其余的字符是可见字符，它们的 ASCII 码值是 32～126，属于数字、字母、各种符号。

对于计算机来说，任何字符都是用 ASCII 码表示的，人要是与计算机交流，虽然可以直接输入字符的 ASCII 码，但这太不人道了，计算机的发明是为了给人解决问题而并非制造问题。人习惯用所见即所得的方式使用字符，我要输入字符 a 的时候，直接按下键盘上的 a 键就行了，不要让我输入其 ASCII 码 0x61。这要求是合理的，我们在键盘上键入的每个按键，都会由输入系统根据 ASCII 码表转换成对应的二进制 ACSII 码形式。这对普通用户来说够用了，他们很少写程序，可是作为程序员，我们经常要输出字符串，字符串中的可见字符直接从键盘敲入就行了，对于那些不可见字符，如回车换行符等，肯定不能用键盘在字符串中直接敲下一个回车键。

我们的问题是不可见字符如何写出来，也就是说我们在写字符串时，如何在其中加入不可见的控制符，这就需要编译器或解释器的支持了。

由于可见字符本身是看得见的，所见即所得，大家在使用中并不会有陌生感。对于那些不可见的控制字符，如果想使用它们时，该怎样表示它们呢？比如我就是要让程序输出一段话，在结束处换行。控制字符看不见摸不着，怎么写出来？所以在使用这些不可见字符时必须想办法让其可见，但又不能表示成其他可见字符，所以，只能让可见字符不表示自身了，哈哈，有点难是吗？这么艰巨的任务显然只用一个可见字符是不可能完成的，于是编译器想出了一个办法，它引用了另一个可见字符'\'来搭配其他可见字符，用这种可见字符组合的形式表达不可见字符。表面上看，字符'\'是让其他可见字符的意义变了，所以称'\'为转义字符，但本质上，这两个可见字符合起来才是完整的不可见字符，比如换行符'\n'，'\'和'n'放到一起才是换行符的意义，并不是因为'n'前面有个'\'，'n'就不再是'n'，而是换行符，一定要清楚不是这样的。

ASCII 码表中任何字符都是 1 个字节大小，在字符串中不可见字符虽然用“转义字符+可见字符”两个字符来表示，但这只是编译器为了让人们能写出不可见字符的方式，目的是让不可见字符变得“可见”，针对的是人，这样人们写程序时就能在字符串中用到不可见字符。不可见字符本身在编译后还是那 1 个字节的 ASCII 码。说白了，我们能够将不可见字符显示出来，原因就是编译器在给我们做支持，它将“转义字符+可见字符”这种形式的不可见字符转换成了该不可见字符的 ASCII 码。

为了说清楚，咱们以编译器为界限，在编译器左边的是人，这里的字符串是供人使用的，转义字符是存在于这一边的。编译器右边的是机器，这里的字符串使用的都是 ASCII 码。

在编译器左边：

```
char* ptr=” abc\n”;
```

此部分对应的内容是 0x61 0x62 0x63 0x5c 0x6e。

编译器右边：

“abc\n”对应的内容是 0x61 0x62 0x63 0xa

编译器的左边和右边是不一样的，区别是对“\n”的处理。编译器左边把它当成了两个字符，编译器右边把它当成了一个字符。想想也是，毕竟代码只是文本字符串，字符串”abc\n”中的'\'和'n'肯定是两个字符，编译器会把'\'和'n'组合到一起成为'\n'而解释成回车换行。可能您还是觉得怀疑，那我说一下编译器对字符串的解释过程。

编译器对字符串的处理一般是逐个字符处理的，这样便于处理转义字符。若发现字符为'\'，就意识到这是转义字符，按常理说后面肯定要跟着另一可见字符，于是先不做任何处理，马上把后面的字符读进来，分析这两个字符的组合是哪个控制字符后一并处理。

咱们这里拿编译器解释字符串”abc\n”举例。

代码中的'\n'本身由两个字符'\'和'n'组成，'\n'是给人看的，用于在字符串中使用，其 ASCII 码是 0xa，是给机器看的。在计算机中，所有的字符都已经成了 ASCII 码，字符串”abc\n”则变成了 ASCII 码：0x61 0x62 0x63 0x5c 0x6e。

编译器要逐个对比字符串中每个字符，前几个字符是'a'、'b'、'c'，这都是可见字符，没有异议，直接处理。当发现字符是'\'，知道这是转义字符，得知道'\'后面的字符是什么才能确定是哪个不可见字符，于是暂停处理'\'，把后面的字符读进来，发现是'n'，便知道这是'\n'，表示一个换行符，于是将'\'和'n'用换行符的 ASCII 代替，原来字符串”abc\n”的 ASCII 码就变成了 0x61 0x62 0x63 0xa。

说得足够多了，我也嫌自己啰嗦了，大家看以下的例子吧，就在图 0-20 中全部解释清楚了。

代码 ASCII.c 过于简单，纯粹是为演示。大家可能注意到了 xxd.sh 这个脚本，它就是 xxd 命令的封装，xxd 命令可以逐字节查看文件，xxd.sh 脚本内容如下。

```
#usage: sh xxd.sh 文件起始地址长度
xxd -u -a -g 1 -s $2 -l $3 $1
#以下为参数解释。
#-u  use upper case hex letters. Default is lower case.
#
```

```
#-a | -autoskip
#        toggle autoskip: A single '*' replaces nul-lines.  Default off.
#
#-g bytes | -groupsize bytes
#     separate the output of every <bytes> bytes (two hex characters or eight bit-digits each)
by a whitespace.  Specify -g 0 to
#     suppress grouping.  <Bytes> defaults to 2 in normal mode and 1 in bits mode.  Grouping does
not  apply  to  postscript  or
#     include style.
#
#-c cols | -cols cols
#                format <cols> octets per line. Default 16 (-i: 12, -ps: 30, -b: 6). Max 256.
#
#-s [+][-]seek
#    start at <seek> bytes abs. (or rel.) infile offset.  + indicates that the seek is relative
to the current stdin file position
#    (meaningless when not reading from stdin).  - indicates that the seek should be that many
characters from the end of
#    the input (or if combined with +: before the current stdin file position).
#    Without -s option, xxd starts at  the  current  file position.
```

```
[work@localhost tmp]$ cat -n ascii.c
     1  char* ptr="abc\n";
     2  void main(){
     3  }
[work@localhost tmp]$ gcc -c -o ascii.o ascii.c
[work@localhost tmp]$ ll ascii.o
-rw-rw-r--. 1 work work 824 10月  24 14:37 ascii.o
[work@localhost tmp]$ ll ascii.c
-rw-rw-r--. 1 work work 34 10月  24 14:34 ascii.c
[work@localhost tmp]$ sh ~/tool/xxd.sh ascii.c 0 34
0000000: 63 68 61 72 2A 20 70 74 72 3D 22 61 62 63 5C 6E  char* ptr="abc\n
0000010: 22 3B 0A 76 6F 69 64 20 6D 61 69 6E 28 29 7B 0A  ";.void main(){.
0000020: 7D 0A                                            }.
[work@localhost tmp]$
[work@localhost tmp]$ ll ascii.o
-rw-rw-r--. 1 work work 824 10月  24 14:37 ascii.o
[work@localhost tmp]$ sh ~/tool/xxd.sh ascii.o 0 824|grep 61
0000040: 61 62 63 0A 00 00 47 43 43 3A 20 28 47 4E 55 29  abc...GCC: (GNU)
0000060: 28 52 65 64 20 48 61 74 20 34 2E 34 2E 36 2D 34  (Red Hat 4.4.6-4
0000070: 29 00 00 2E 73 79 6D 74 61 62 00 2E 73 74 72 74  )...symtab..strt
0000080: 61 62 00 2E 73 68 73 74 72 74 61 62 00 2E 74 65  ab..shstrtab..te
0000090: 78 74 00 2E 72 65 6C 2E 64 61 74 61 00 2E 62 73  xt..rel.data..bs
00000a0: 73 00 2E 72 6F 64 61 74 61 00 2E 63 6F 6D 6D 65  s..rodata..comme
00000b0: 6E 74 00 2E 6E 6F 74 65 2E 47 4E 55 2D 73 74 61  nt..note.GNU-sta
0000310: 00 00 00 00 05 00 00 00 12 00 01 00 00 61 73 63  .............asc
0000320: 69 69 2E 63 00 70 74 72 00 6D 61 69 6E 00 00 00  ii.c.ptr.main...
[work@localhost tmp]$
```


▲图 0-20　查看编译后的转义字符

希望对大家理解转义字符有帮助。

0.28 MBR、EBR、DBR 和 OBR 各是什么

这几个概念主要是围绕计算机系统的控制权交接展开的，整个交接过程就是个接力赛，咱们从头梳理。

计算机在接电之后运行的是基本输入输出系统 BIOS，大伙儿知道，BIOS 是位于主板上的一个小程序，其所在的空间有限，代码量较少，功能受限，因此它不可能一人扛下所有的任务需求，也就是肯定不能充当操作系统的角色（比如说让 BIOS 运行 QQ 是不可能的），必须采取控制权接力的方式，一步步地让处理器执行更为复杂强大的指令，最终把处理器的使用权交给操作系统，这才让计算机走上了正轨，从而可以完成各种复杂的功能，方便人们的工作和生活。采用接力式控制权交接，BIOS 只完成一些简单的检测或初始化工作，然后找机会把处理器使用权交出去。交给谁呢？下一个接力棒的选手是 MBR，为了方便 BIOS 找到 MBR，MBR 必须在固定的位置等待，因此 MBR 位于整个硬盘最开始的扇区。

MBR 是主引导记录，Master 或 Main Boot Record，它存在于整个硬盘最开始的那个扇区，即 0 盘 0 道 1 扇区，这个扇区便称为 MBR 引导扇区。注意这里用 CHS 方式表示 MBR 引导扇区的地址，因此扇区地址以 1 开始，顺便说一句，LBA 方式是以 0 为起始为扇区编址的，有关 CHS 和 LBA 的内容会在后面章节介绍。一般情况下扇区大小是 512 字节，但大伙儿不要把这个当真理，有的硬盘扇区并不是 512 字节。

在 MBR 引导扇区中的内容是：

（1）446 字节的引导程序及参数；

（2）64 字节的分区表；

（3）2 字节结束标记 0x55 和 0xaa。

在 MBR 引导扇区中存储引导程序，为的是从 BIOS 手中接过系统的控制权，也就是处理器的使用权。任何一棒的接力都是由上一棒跳到下一棒，也就是上一棒得知道下一棒在哪里才能跳过去，否则权利还是交不出去。BIOS 知道 MBR 在 0 盘 0 道 1 扇区，这是约定好的，因此它会将 0 盘 0 道 1 扇区中的 MBR 引导程序加载到物理地址 0x7c00，然后跳过去执行，这样 BIOS 就把处理器使用权移交给 MBR 了。

既然 MBR 称为“主”引导程序，有“主”就得有“次”， MBR 的作用相当于下一棒的引导程序总入口，BIOS 把控制权交给 MBR 就行了，由 MBR 从众多可能的接力选手中挑出合适的人选并交出系统控制权，这个过程就是由“主引导程序”去找“次引导程序”，这么说的意思是“次引导程序”不止一个。也许您会问，为什么 BIOS 不直接把控制权交给“次引导程序”？原因是 BIOS 受限于其主板上的存储空间，代码量有限，本身的工作还做不过来呢，因此心有余而力不足。好啦，下面开始下一轮的系统控制权接力。不要忘了，MBR 引导扇区中除了引导程序外，还有 64 字节大小的分区表，里面是分区信息。分区表中每个分区表项占 16 字节，因此 MBR 分区表中可容纳 4 个分区，这 4 个分区就是“次引导程序”的候选人群，MBR 引导程序开始遍历这 4 个分区，想找到合适的人选并把系统控制权交给他。

通常情况下这个“次引导程序”就是操作系统提供的加载器，因此 MBR 引导程序的任务就是把控制权交给操作系统加载器，由该加载器完成操作系统的自举，最终使控制权交付给操作系统内核。但是各分区都有可能存在操作系统，MBR 也不知道操作系统在哪里，它甚至不知道分区上的二进制 01 串是指令，还是普通数据，好吧，它根本分不清楚上面的是什么，谈何权利交接呢。

为了让 MBR 知道哪里有操作系统，我们在分区时，如果想在某个分区中安装操作系统，就用分区工具将该分区设置为活动分区，设置活动分区的本质就是把分区表中该分区对应的分区表项中的活动标记为 0x80。MBR 知道“活动分区”意味着该分区中存在操作系统，这也是约定好的。活动分区标记位于分区表项中最开始的 1 字节（有关分区内容，后面介绍分区的章节中会细说），其值要么为 0x80，要么为 0，其他值都是非法的。0x80 表示此分区上有引导程序，0 表示没引导程序，该分区不可引导。MBR 在分析分区表时通过辨识“活动分区”的标记 0x80 开始找活动分区，如果找到了，就将 CPU 使用权交给此分区上的引导程序，此引导程序通常是内核加载器，下面就直接以它为例。

“控制权交接”是处理器从“上一棒选手”跳到“下一棒选手”来完成的，内核加载器的入口地址是这里所说的“下一棒选手”，但是内核加载器在哪里呢？虽然分区那么大，但 MBR 最想去看的是内核加载器，不想盲目地看看。因此您想到了，为了 MBR 方便找到活动分区上的内核加载器，内核加载器的入口地址也必须在固定的位置，这个位置就是各分区最开始的扇区，这也是约定好的。这个“各分区起始的扇区”中存放的是操作系统引导程序——内核加载器，因此该扇区称为操作系统引导扇区，其中的引导程序（内核加载器）称为操作系统引导记录 OBR，即 OS Boot Record，此扇区也称为 OBR 引导扇区。在 OBR 扇区的前 3 个字节存放了跳转指令，这同样是约定，因此 MBR 找到活动分区后，就大胆主动跳到活动分区 OBR 引导扇区的起始处，该起始处的跳转指令马上将处理器带入操作系统引导程序，从此 MBR 完成了交接工作，以后便是内核的天下了。

不过 OBR 中开头的跳转指令跳往的目标地址并不固定，这是由所创建的文件系统决定的，对于 FAT32 文件系统来说，此跳转指令会跳转到本扇区偏移 0x5A 字节的操作系统引导程序处。不管跳转目标地址是多少，总之那里通常是操作系统的内核加载器。

计算机历史中向来把兼容性放在首位，这才是计算机蒸蒸日上的原因。OBR 是从 DBR 遗留下来的，要想了解 OBR，还是先从了解 DBR 开始。DBR 是 DOS Boot Record，也就是 DOS 操作系统的引导记录（程序），DBR 中的内容大概是：

（1）跳转指令，使 MBR 跳转到引导代码；

（2）厂商信息、DOS 版本信息；

（3）BIOS 参数块 BPB，即 BIOS Parameter Block；

（4）操作系统引导程序；

（5）结束标记 0x55 和 0xaa。

在 DOS 时代只有 4 个分区，不存在扩展分区，这 4 个分区都相当于主分区，所以各主分区最开始的扇区称为 DBR 引导扇区。后来有了扩展分区之后，无论分区是主分区，还是逻辑分区，为了兼容，分区最开始的扇区都作为 DOS 引导扇区。但是其他操作系统如 UNIX，Linux 等为了兼容 MBR 也传承了这个习俗，都将各分区最开始的扇区作为自己的引导扇区，在里面存放自己操作系统的引导程序。由于现在这个“分区最开始的扇区”引导的操作系统类型太多了，而且 DOS 还退出历史舞台了，所以 DBR 也称为 OBR。

这里提到了扩展分区就不得不提到 EBR。当初为了解决分区数量限制的问题才有了扩展分区，EBR 是扩展分区中为了兼容 MBR 才提出的概念，主要是兼容 MBR 中的分区表。分区是用分区表来描述的，MBR 中有分区表，扩展分区中的是一个个的逻辑分区，因此扩展分区中也要有分区表，为扩展分区存储分区表的扇区称为 EBR，即 Expand Boot Record，从名字上看就知道它是为了“兼容”而“扩展”出来的结构，兼容的内容是分区表，因此它与 MBR 结构相同，只是位置不同，EBR 位于各子扩展分区中最开始的扇区（注意，各主分区和各逻辑分区中最开始的扇区是操作系统引导扇区），理论上 MBR 只有 1 个，EBR 有无数个。有关扩展分区的内容还是要参见后面有关分区的章节，那里介绍得更细致。

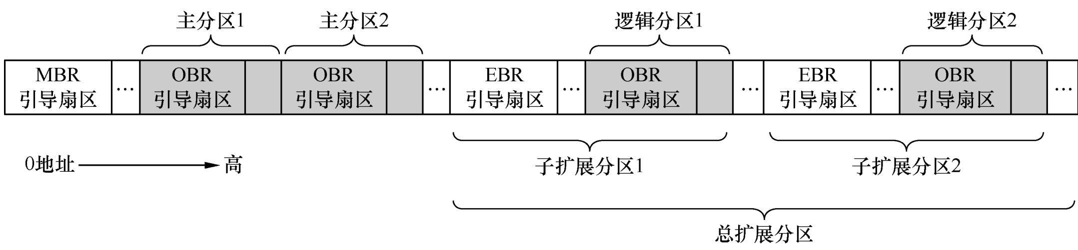
现在总结一下。

EBR 与 MBR 结构相同，但位置和数量都不同，整个硬盘只有 1 个 MBR，其位于整个硬盘最开始的扇区——0 道 0 道 1 扇区。而 EBR 可有无数个，具体位置取决于扩展分区的分配情况，总之是位于各子扩展分区最开始的扇区，如果此处不明白子扩展分区是什么，到了以后跟踪分区的章节中大伙儿就会明白。OBR 其实就是 DBR，指的都是操作系统引导程序，位于各分区（主分区或逻辑分区）最开始的扇区，访扇区称为操作系统引导扇区，即 OBR 引导扇区。OBR 的数量与分区数有关，等于主分区数加逻辑分区数之和，友情提示：一个子扩展分区中只包含 1 个逻辑分区。

MBR 和 EBR 是分区工具创建维护的，不属于操作系统管理的范围，因此操作系统不可以往里面写东西，注意这里所说的是“不可以”，其实操作系统是有能力读写任何地址的，只是如果这样做的话会破坏“系统控制权接力赛”所使用的数据，下次开机后就无法启动了。OBR 是各分区（主分区或逻辑分区）最开始的扇区，因此属于操作系统管理。

DBR、OBR、MBR、EBR 都包含引导程序，因此它们都称为引导扇区，只要该扇区中存在可执行的程序，该扇区就是可引导扇区。若该扇区位于整个硬盘最开始的扇区，并且以 0x55 和 0xaa 结束，BIOS 就认为该扇区中存在 MBR，该扇区就是 MBR 引导扇区。若该扇区位于各分区最开始的扇区，并且以 0x55 和 0xaa 结束，MBR 就认为该扇区中有操作系统引导程序 OBR，该扇区就是 OBR 引导扇区。

DBR、OBR、MBR、EBR 结构中都有引导代码和结束标记 0x55 和 0xaa，因此很多同学都容易把它们搞混。不过它们最大的区别是分区表只在 MBR 和 EBR 中存在，DBR 或 OBR 中绝对没有分区表。MBR、EBR、OBR 的位置关系如图 0-21 所示。



▲图 0-21　MBR、EBR、OBR 位置关系

您看，MBR 位于整个硬盘最开始的块，EBR 位于每个子扩展分区，各子扩展分区中只有一个逻辑分区。MBR 和 EBR 位于分区之外的扇区，而 OBR 则属于主分区和逻辑分区最开始的扇区，每个主分区和逻辑分区中都有 OBR 引导扇区。有关分区更详细的内容请参阅后面跟踪分区表的章节。

第 1 章　部署工作环境

1.1 工欲善其事，必先利其器

如果您觉得操作系统已属于很底层的东西，我双手赞成。但是如果您像我之前一样，觉得底层的东西无法用上层高级的东西来构建，现在可以睁大眼睛好好看看下面要介绍的东西了。

首先，操作系统是软件。软件是由编程语言来实现的，即使是编译器本身，它的开发人员都不愿意用底层语言去构建（GCC 是用 C 语言完成的），只有到万不得已的时候才会用汇编语言来写。我们也是一样，能用省事的方法就不要自找麻烦，如果某位大神能直接写机器码，小弟真心恳求与您见上一面，希望您收我为徒，我要当面磕头拜师。不过话又说回来了，直接写机器码也并不是什么明智的做法，毕竟费力不讨好，不过毅力还是值得钦佩的。同学们不要被我虔诚的态度误解为直接写机器码是不可能的事，这个能，必须能，写汇编编译器的同学做的就是这样的事，原则上只要按照 IA-32 指令格式往二进制文件中写指令，就一定能让 CPU 理解自己，能够直接同 CPU 对话了……停，赶紧回来，咱们是来写操作系统的，赶紧进入主题。

1.2 我们需要哪些编译器

C 语言虽然不是为设计大型软件而生的，但其却被用来开发大型软件。

现代操作系统基本上是用 C 语言再结合汇编语言开发的，所以 C 语言编译器，我们选择的是 gcc。而汇编语言编译器，我们选择的是 nasm。为什么选择这两个，首先因为它们都是开源软件，其次其强大的功能不亚于同类的商业软件。

1.2.1 世界顶级编译器 GCC

秉着简单至上的原则，我们在开发过程中，能用简单的工具就不用复杂的。所以我们的系统，绝大部分是 C 语言实现的，而且并不需要多么高深的算法及数据结构功底。

另外我们在 Linux 下开发，所以首先的编译器就是 GCC，基本上没有人不了解这个大名鼎鼎的开源编译器了。出于对这个编译器的膜拜，我还是引用 wiki 上的介绍：

GNU 编译器套装（GNU Compiler Collection，GCC），是一套由 GNU 开发的编程语言编译器。它是一套以 GPL 及 LGPL 许可证所发行的自由软体，也是 GNU 计划的关键部分，亦是自由的类 Unix 及苹果电脑 Mac OS X 操作系统的标准编译器。GCC（特别是其中的 C 语言编译器）也常被认为是跨平台编译器的标准。

GCC 是由理查德・马修・斯托曼在 1985 年开始的。他首先扩展一个旧有的编译器，使它能编译 C，这个编译器一开始是以 Pastel 语言所写的。Pastel 是一个不可移植的 Pascal 语言特殊版，这个编译器也只能编译 Pastel 语言。为了让自由软件有一个编译器，后来此编译器由斯托曼和 Len Tower 在 1987 年以 C 语言重写并成为 GNU 专案的编译器。GCC 的建立者由自由软件基金会直接管理。

GCC 原名为 GNU C 语言编译器（GNU C Compiler），因为它原本只能处理 C 语言。GCC 很快扩展，以 2011 年 10 月 26 日释出的 4.6.2 版为准，可处理的编程语言有：

```
1.Ada(GNAT)
2.C(GCC)
3.C++(G++)
4.Fortran(Fortran 77: G77,Fortran 90: GFORTRAN)
```

```
5.Java(编译器:GCJ;解释器:GIJ)
6.Objective-C(GOBJC)
7.Objective-C++
8. Go
```

好啦，介绍结束，看上去 GCC 很厉害，居然可以支持这么多语言。不愧是出自理查德·马修·斯托曼（Richard Matthew Stallman）之手，只要学过计算机的读者便了解此人，他到底有多厉害呢，看网友对他的评价："曾独自一人与一众 lisp 黑客高手进行比赛……"好了，多说已无益，简单的半句话便彻底表达了此人深厚的计算机功力。

回到正题，Linux 系统会自带 GCC，如果您的发行版中没有，可以到网站 http：//gcc.gnu.org/下载。

1.2.2 汇编语言编译器新贵 NASM

新是相对于旧来说的，旧的汇编器 MASM 和 TASM 已经过时了，从名称上可以看出字母 n 是在 m 之后，其功能必然有所超越才会被大家接受。

请用一句话概括 NASM 优势在哪里。免费+语法简洁使人舒适+支持 Linux 平台。这里所说的任何一个理由都是其他同类产品不具备的，敏锐的同学是不是察觉到了什么……哈哈，怎么给人的感觉是：其他编译器不是花钱，就是语法怪异让人不爽，要么就不支持 Linux，看上去选择 nasm 是没得可选了？我就不自问自答了，反正 NASM 语法很接近咱们当初学的 Intel 语法，我是用得很爽呢。这里就不再比较其优越性了，大家若感兴趣还是自行查阅吧。

同样是为了抒发一下对这位新贵的"爱慕之情"，简要介绍还是很有必要的。

NASM 是一个为可移植性与模块化而设计的一个 80x86 的汇编器。它支持相当多的目标文件格式，包括 Linux 和'NetBSD/FreeBSD'，'a.out'，'ELF'，'COFF'，微软 16 位的'OBJ'和'Win32'。它还可以输出纯二进制文件。它的语法设计得相当的简洁易懂，和 Intel 语法相似但更简单。它支持'Pentium'，'P6'，'MMX'，'3DNow!'，'SSE'和'SSE2'指令集，

介绍完了之后，咱们讨论下为什么要用汇编语言开发系统呢？就目前来看，无论再怎么要求开发过程简单，也避免不了用汇编语言，尤其是开发操作系统这类底层软件。越底层的软件就越要与硬件直接打交道，这就要求在语言层面上给开发人员提供访问端口寄存器的方法。显然，目前的高级语言都做不到这一点，像 C 语言这类偏底层的语言都不支持修改寄存器，用汇编语言则是不可避免的事了。

包括我在内的很多同学一听要用汇编了，都有一种小小的恐惧感，认为这是一种不好掌握的东西（我没有称之为语言而是称之为东西，是因为曾经有个女同学都不知道汇编是什么），而且程序编写起来特别麻烦，要考虑的东西太多了，代码逻辑写起来不够直接，似乎总是在迂回……以至于我们经常被汇编语言"搞定"。我个人的感觉是当我熟悉了汇编语言后，甚至觉得有一点亲切呢。当然了，任何陌生的事物经过熟悉的过程后都会变得有亲切感，关键是咱们得扛到对它熟悉为止，不能让心里的畏惧战胜自己。用汇编语言和 CPU 直接对话，想想就有点小兴奋呢。

不过好在我们需要用汇编的地方只是一些硬件访问、中断调用、端口读写、线程切换之类（怎么看上去好多……），我们可以写出一些通用的代码来减少汇编的枯燥。总之，只有不得不用时我们才会向汇编语言屈服。

1.3 操作系统的宿主环境

操作系统虽然是软件，但其可不是一般的软件。我们平时写出来的程序都是基于操作系统之上的，程序本身是由操作支持的，开发人员只要专注于自己这块业务逻辑就好了，很多复杂的问题是不需要开发人员考虑的。而操作系统这个软件靠谁来支持呢？是靠你自己……这是用一身老骨头扛出来的，现在明白为什么 Linux 之父 Linus 那么强壮了吧，不是谁都能随随便便成功的，所以，写操作系统那可是要有个好身板，要多锻炼身体才能熬得住，看完这章赶紧出去跑步吧，玩笑玩笑。如果一般应用软件能称得上鱼香肉丝的话，那操作系统得相当于龙虾鲍鱼，这可是硬菜，不出去跑个几公里都啃不下来呢。哈哈，其实也没

那么夸张，现在有很多计算机大牛写了好多开源软件帮助我们调试操作系统。话说，自从有了虚拟机，我再也用不着锻炼身体了，每次出现 bug 时不需要重启真机了，只需要重启虚拟机就好。

1.3.1　什么是虚拟机

虚拟机在当今已经不是陌生的概念了，要是在几年前，我还得搬出个概念放在这给大家看看。个人觉得，要解释一个东西是什么，不如直接解释这个东西解决了哪些问题，这样大家自然就从本质上真正理解了它是什么。

没有虚拟机的时候，一台机器只有交给一个用户使用，而且一个人根本无法将这台机器的性能完全发挥出来，造成了极大的浪费不算，还有很多人正等着用呢。于是出现了虚拟机的需求：将一台物理机通过软件逻辑分割成几个虚拟的计算机，每个计算机之间互不干涉，即使一台虚拟计算机崩溃了也只是影响了它自身，不会让整个物理机瘫痪，安全可靠，可以自由测试而不必担心损伤物理机。这不仅在硬件投入上节省了大笔开销，还让更多的人同时使用了计算机资源。

现在很多厂商都在搞虚拟化，如域名的虚拟空间，还有如火如荼的阿里云，这是虚拟机的应用。虚拟机就是用软件来模拟硬件。虚拟机只是一个普通的进程，该进程模拟了硬件资源，在虚拟机中运行的程序其所做出的任何行为都先被虚拟机检查，由虚拟机分析后，代为向操作系统申请。

上面对虚拟机的解释是主观上的理解，我可不愿意说概念了，因为概念是对事物的抽象。抽象就意味着不易理解，容易把简单的事复杂化。我举个简单的例子来说明什么是虚拟机。

假设 V 是虚拟机进程，U 是普通的用户程序。程序运行起来才叫进程，进程是要有 pcb 的，程序老实地放在磁盘上不动，那可不叫进程。虚拟机跑起来后，就形成了进程 V，在它被调度期间，CPU 执行的是此进程中的指令。让虚拟机执行 U 程序，有如解释器进程在解析脚本文件一样，此时的 U 程序被当作参数传给了 V 程序，U 程序就像文章一样由 V 进程阅读。还是拿解释型语言举例子，比如 python 语言，其脚本从来就没有直接作用于 CPU 上，而是将其字节码交给了 python 解释器，这个解释器将通过 python 虚拟机来代为完成 python 脚本中的代码行为。

让我们说得再具体一点，比如在 Linux 平台上，写了一个 python 脚本文件 file.py，其中有这样一句代码：fh=open（“hello.txt”，'w'），这是在用可写的方式打开 hello.txt 文件，将其句柄返回给 fh。自此操作文件句柄 fh 便操作了文件 hello.txt。python 虚拟机是一个进程，它是直接作用在硬件之上的。当它分析 python 脚本 file.py 中上面的那句代码时，发现有关键字 open（当然关键字得是 python 解释器支持的，此解释器为它们而生才行），于是执行了 open（“hello.txt”，'w'）函数，其内部是封装的系统调用（系统调用这方面内容以后咱们在自己的系统中细说），通过系统调用，python 虚拟机替 python 脚本完成了打开 hello.txt 的工作。

选择虚拟机的原因如下。

1．运行方便

它在宿主系统上只是一个进程，在宿主系统如 Linux 眼里，它与一般的用户进程是没任何区别的。进程咱们都可以随意启动，虚拟机也是一样的，在这一点保证了使用上的方便性。

2．保护计算机

如果您有一般的软件开发经验，就会了解，很少有程序能一下就编译通过。当然，如果您的编程经验无比丰富，代码无比规范，无比了解编译器，确实不需要虚拟机来调试了，编写完成后直接就能运行。以上我用了三个“无比”，打造了似乎没有人能达到这种水平的假象，其实是有的。不知道大家听说过 Jon Skeet 没有，他是谷歌软件工程师，《C# In Depth》就是他的作品。看看别人对他是怎样评价的，看完之后您就知道我说的并不夸张了。

“他并不需要调试器，只要他盯着代码看几眼，Bug 自己就跑出来了”。

“他根本不需要什么编程规范，他的代码就是规范”。

“如果他的代码没有通过编译，编译器厂商就会道歉”。

如果咱们都不能保证写出这样质量出色的代码，咱们还是老老实实地装虚拟机吧。因为如果要把操作

系统装在真机器上，每次调试的时候，无论代码是否崩溃，都是要重启计算机的。为了保护咱们的爱机，虚拟机必装不可。不知道你们心疼电脑吗，反正我要是一天开电脑三次以上，我就会很自责，不知道这种性格和水瓶座有没有关系。

说了虚拟机的好处，那咱们有哪些虚拟机可用呢。一般的有 qemu、bochs、virtualBox、xen 和 vmware 等。

大家都这么忙，能有时间拿起书不容易啊，为了不浪费您宝贵的时间，我就不说和本书不相关的虚拟机了，种类再多，也只是选一个。我们要用的就是 bochs。选择 bochs 的理由如下。

（1）开源，有感于作者的奉献精神，我们要支持作者（当然 qemu 也是）。

（2）支持调试，不仅原生支持调试，还支持 gdb 远程调试（当然 qemu 也是）。

（3）我只会这个。

对于虚拟机的选择，能工作能调试够用就行了，遇到问题时再寻求新方案也不迟，毕竟咱们的重点是后面的写操作系统，学太多的虚拟机也没啥用。

介绍一下 bochs 吧，怎么也得让大家有个初步的印象。下面的内容是我从维基百科翻译过来的，其实就是从繁体中文翻译成了简体中文，而且只有几个繁体字，哈哈。

Bochs（发音：box）是一个以 LGPL 许可证发放的开放源代码的 x86、x86-64IBM PC 兼容机模拟器和调试工具。它支持处理器（包括保护模式）、内存、硬盘、显示器、以太网、BIOS、IBM PC 兼容机的常见硬件外设的仿真。

许多客户操作系统能通过该仿真器运行，包括 DOS、Microsoft Windows 的一些版本、AmigaOS 4、BSD、Linux、MorphOS、Xenix 和 Rhapsody（Mac OS X 的前身）。Bochs 能在许多主机操作系统运行，例如 Windows、Windows Mobile、Linux、Mac OS X、iOS 和 PlayStation 2。

Bochs 主要用于操作系统开发（当一个模拟操作系统崩溃，它不崩溃主机操作系统，所以可以调试仿真操作系统）和在主机操作系统运行其他来宾操作系统。它也可以用来运行不兼容的旧的软件（如电脑游戏）。

它的优点在于能够模拟跟主机不同的机种，例如在 Sparc 系统里模拟 x86，但缺点是它的速度却慢得多。

介绍完了，不知道您看了吗，不看也行，反正以后咱们实际应用时还会细说的。

1.3.2 盗梦空间般的开发环境，虚拟机中再装一个虚拟机

很多同学电脑的系统都是 Windows，个别的是 Mac OS，还有的同学用的是 Linux。作为一名 Linux 粉丝，我的开发环境必然建立在 Linux 平台下。那对于其他系统的用户，你们可以自己部署相应平台的开发环境，用的工具都是一样的，无非是换个相应平台的版本。不过我担心由于平台不同而造成这样那样的问题，会减退大家学习的积极性，毅力不足的同学还没开始写操作系统就急流勇退了，我后面精彩的内容没有观众该怎么办。为了减少学习的困难，也为了让大家继续为我后面的内容捧场，我在睡觉的时候想到一个好办法，能让大家的开发环境极大限度地得到统一，就是我们加一层，再安装一个虚拟机。

首先虚拟机是个软件，不会伤害咱们的爱机。我一度认为虚拟机是一项非常伟大的发明，甚至认为它是给像我这样的穷人最好的礼物，当初学习思科（网络、路由器等方面知识）时，幸亏有虚拟机来模拟多台计算机，否则还真买不起第二台电脑，所以大家一定要好好学习，不要像我当初那样辜负了虚拟机。我们的方案是这个虚拟机就用 virtualBox 吧，虽然在结尾加了个“吧”，但我丝毫没有征求大家意见的意思，哈哈，抱歉，我这也绝对不是强硬。让小弟我给大家个交待。

（1）个人觉得 virtualBox 比 vmware 更轻量，配置起来更简单。

（2）virtualBox 是免费的，不需要破解，这一点很重要。

（3）因为我不想改成别的了，嫌麻烦，请大家原谅。

交待过了之后，大家还是根据自己喜好选择虚拟机，大家觉得哪个方便就用哪个。

方案再多也总该选择一个，我选择的方案是在 virtualBox 中安装个操作系统。因为要在 Linux 下开发，所以选的是与 redhat 很接近的 CentOS，我用的版本是 6.3，本书中以后便以 virtualBox+CentOS 6.3 为例。

由于我后续的环境部署都是在此版本上进行的，没遇到什么大问题，确实感觉很稳定，简单可依赖。

在 CentOS 中再装个 bochs，最终我们的代码运行在 bochs 中。

想当初莱欧纳多的电影《盗梦空间》上映时，很多朋友都被故事的新颖所吸引，大概意思是通过潜入别人的梦中去窃取机密，如果第一层梦境窃取不到，还可以在梦中继续睡觉，再进入第二层，也就是梦中梦。这就是标题中所说的，虚拟机中再装个虚拟机，您看，描述还是有些形象的。

1.3.3　virtualBox 下载，安装

virtualBox 官方下载地址是 http：//download.virtualbox.org/virtualbox，大家选择一个适合自己系统平台的版本，我安装的是 4.2.12 mac 版本，具体下载地址是 http：//download.virtualbox.org/virtualbox/4.2.12/，大家可以自行选择。

MacOS 和 Windows 基本上 virtualBox 的安装是一路回车，没什么可说的。如果您用的系统是 Linux，我更觉得说什么都显得多余，因为能用 Linux 办公，说明您完全有能力安装成功。

1.3.4　Linux 发行版下载

可以在 mirrors.163.com 这个国内的镜像源去下载自己喜欢的版本。

由于 CentOS 官方的告示，目前 6 系列的版本只有 6.5 可用，其他低版本不再支持。喜欢 CentOS 的朋友可以趁机装个新版本。如果您像我一样执拗非 6.3 版本不装，我也给出了官方的链接地址，大家斟酌安装。

http：//vault.centos.org/6.3/isos/i386/CentOS-6.3-i386-bin-DVD1.iso

大家根据自己虚拟机的种类开始安装 Linux 吧，由于版本和宿主系统种类较多我不便将安装步骤一一给出，大家若有不懂的问题请自行查阅，百度经验上有很多方法，若第一次用虚拟机，大家可以参考下面链接的方法。

http：//jingyan.baidu.com/article/414eccf61d12cc6b431f0ae7.html。

1.3.5　Bochs 下载安装

在完成了 Linux 发行版的安装后，现在到了安装 bochs 的环节，这是我们的操作系统最终的宿主机。

由于我的工作是运维，所以练就了任何软件包都要从源码安装的“陋习”，从来不信任任何软件包。因为只有从源码安装的版本才会在其配置和编译过程中根据所在的平台的特性去优化，这些是其他形式的软件包不可比拟的。举个例子，将别人的 Windows 系统直接 ghost 到自己的机器上和从光盘安装 Windows 比，哪个装的 Windows 系统用得更稳定，哪个安装方法能让 Windows 坚持到半年才重装一次……我不能再说了，我作为 Linux 粉丝的事实已表露无遗。虽然我个人偏爱 Linux，但绝对不能否认，是 Windows 把我带入计算机世界的。这个 IT 世界若没有 Windows 将暗淡 70%的光芒。其实原先我写的是 90%，我怕有人问我这个数是怎么来的，这是我一拍脑门随口说出来的，所以我稳妥起见改为了 70%，总之，不能无视 Windows 的伟大功绩。

bochs 的安装相对要麻烦一些，不光是装上去就行了，还需要配置一下。

软件包得传到虚拟机上才能安装到虚拟机里，如何传上去呢。下面建议了 3 个方案。

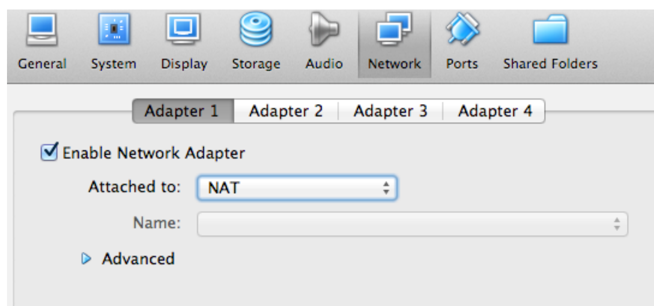
（1）给虚拟机装个 ftp，通过 ftp 上传。

（2）让虚拟机连网，直接下载。

（3）虚拟机支持 USB，通过 U 盘上传软件包。

第 1 个方案需要配置 ftp 服务器，我用的是 proftpd，相对来说有点麻烦，也是需要单独配置的。而且默认 Linux 的 iptables 会有一些规则，需要手动将其关闭。

第 2 个方案较简单，在您的宿主系统可以连网的情况下，需要您自己配置一下 virtualBox 的网卡，将网卡部分改为 NAT 可以通过宿主系统连网，将网卡改为桥接可以直接连网。由于大家的版本不统一，虽然不知道界面是否接近，但菜单名称总该是一样的。我用的是 mac 版 virtual Box，给大家截个图看看，如图 1-1 所示。



▲图 1-1　virtual Box

将网卡模式改为 NAT 后，虚拟机就可以连网了。

第 3 个方案最方便了，大家自己试一下吧。

好了，下面就假设大家能够把安装包上传到虚拟机中，安装走起。

1．下载 bochs

官方地址是 http：//sourceforge.net/projects/bochs/files/bochs/，我安装的版本是 2.6.2，下载后的文件是 bochs-2.6.2.tar.gz。

2．解压压缩包 tar zxvf bochs-2.6.2.tar.gz

3．编译

先进入到目录 cd bochs-2.6.2，开始 configure、make、make install 三步曲。

```
./configure \
--prefix=/your_path/bochs \
--enable-debugger\
--enable-disasm \
--enable-iodebug \
--enable-x86-debugger \
--with-x \
--with-x11
```

注意各行结尾的'\'字符前面有个空格。下面简要说明一下 configure 的参数。

```
--prefix=/your_path/bochs 是用来指定 bochs 的安装目录，根据个人实际情况将 your_path 替换为自己待安装的路径。
--enable-debugger 打开 bochs 自己的调试器。
--enable-disasm 使 bochs 支持反汇编。
--enable-iodebug 启用 io 接口调试器。
--enable-x86-debugger 支持 x86 调试器。
--with-x 使用 x windows。
--with-x11 使用 x11 图形用户接口。
```

上面的编译参数是不支持 gdb 远程调试的，如果想用 gdb 调试，就要将参数--enable-debugger 替换为--enable-gdb-stub。

--enable-gdb-stub 用来打开对 gdb 的支持，这样我们就可以用 gdb 来远程调试了。

不过，需要注意的是不能同时打开这两个开关，否则 bochs 会报错，即 configure：error：--enable-debugger and --enable-gdb-stub are mutually exclusive。

也就是说，bochs 本身是支持调试的，要么用本身的调试功能，要么用 gdb 的调试功能，鱼和熊掌在一台模拟器上不可兼得。我说的是一台模拟器上不可兼得，所以，如果您愿意的话，可以用这两个参数各编译一版，只要--prefix 指向不同的路径就行了，想用哪个就启用哪个。

不过我在开发过程中，只用过不超过 5 次的 gdb 调试，还是习惯 bochs 自己的调试功能，个人觉得它更强大，调试粒度更细微，反而更灵活。个人建议，直接用给出的 configure 参数就行，不要打开--enable-gdb-stub。

configure 之后，会生成 Makefile，可以开始编译了。

```
make
```

若编译时没有问题，就直接执行下面这句。

```
make install
```

完成安装。

补充一下，在编译用 bochs 调试功能的版本时（用--enable-debugger），曾经安装失败过，如果您也在安装过程中失败了，恰好出现类似下面的报错：

```
undefined reference to 'pthread_create'
undefined reference to 'pthread_join'
```

您可以按照下面的方法解决。如果不是这个报错，亲，您可能要辛苦一下自行解决啦。

上面报错的原因：

pthread 库不是 Linux 系统默认的库，连接时需要使用静态库 libpthread.a，所以在使用 pthread_create()创建线程，以及调用 pthread_atfork()函数建立 fork 处理程序时，需要链接该库。

解决方案：

在编译中要加 -lpthread 参数。用 vim 编译 makefile，vim 是 Linux 下功能最为强大的文本编辑器。vim Makefile 回车：

编辑第 92 行，将 thread 库加入，将其放在行末尾就行了。

```
IBS =-lm -lgtk-x11-2.0 -lgdk-x11-2.0 -latk-1.0 -lgio-2.0 -lpangoft2-1.0 -lgdk_pixbuf-2.0 –lpangocairo
-1.0 -lcairo -lpango-1.0 -lfreetype -lfontconfig -lgobject-2.0 -lgmodule-2.0 -lglib-2.0 –lpthread
```

重新编译，make 回车，看问题是否解决，成功解决后直接 make install 回车。

1.4 配置 bochs

安装完成后该配置 bochs 了，它是通过配置文件完成的。

要说这个配置文件，它有点类似 BIOS。我们在开机时按下的 del、esc，或者 F2 键，各个机型进入 BIOS 方式有所不同，但差不多就那几种方式。BIOS 中会显示各种硬件的信息，还有启动顺序等。Bochs 既然是模拟硬件的，它就得知道，您需要它模拟的计算机是什么样的，换句话说，在这个虚拟机中有哪些硬件，启动顺序是什么，是从软盘开始，还是从硬盘开始？人家也得像模像样地跟 BIOS 差不多才行。给 bochs 配置硬件的方法，就是写一个配置文件给它，bochs 启动时会找到此文件，根据文件内容创建自己，这样咱们的虚拟机就健全了。

在安装目录下有样本文件：share/doc/bochs/bochsrc-sample.txt。由于此文件有 1130 行，确实有些长，就不贴出来了，摘点重点内容，关于启动顺序，可参见该文件的以下几行（左列的数字是行号）。

```
…
…
 531 #=================================================
 532 # BOOT:
 533 # This defines the boot sequence. Now you can specify up to 3 boot drives,
 534 # which can be 'floppy', 'disk', 'cdrom' or 'network' (boot ROM).
 535 # Legacy 'a' and 'c' are also supported.
 536 # Examples:
 537 #    boot: floppy
 538 #    boot: cdrom, disk
 539 #    boot: network, disk
 540 #    boot: cdrom, floppy, disk
 541 #================================================
 542 #boot: floppy
 543 boot: disk


下面是能够支持 gdb 的 bochs 配置文件，给大家当作参考。
[work@localhost bochs]$ cat bochsrc.disk  用 cat 命令显示 bochsrc.disk
###############################################
# Configuration file for Bochs
###############################################
```

```
# 第一步，首先设置 Bochs 在运行过程中能够使用的内存，本例为 32MB。
# 关键字为：megs

megs： 32

# 第二步，设置对应真实机器的 BIOS 和 VGA BIOS。
# 对应两个关键字为：romimage 和 vgaromimage

romimage: file=/实际路径/bochs/share/bochs/BIOS-bochs-latest
vgaromimage: file=/实际路径/bochs/share/bochs/VGABIOS-lgpl-latest

# 第三步，设置 Bochs 所使用的磁盘，软盘的关键字为 floppy。
# 若只有一个软盘，则使用 floppya 即可，若有多个，则为 floppya，floppyb…
#floppya: 1_44=a.img, status=inserted

# 第四步，选择启动盘符。
#boot: floppy   #默认从软盘启动，将其注释
boot: disk      #改为从硬盘启动。我们的任何代码都将直接写在硬盘上，所以不会再有读写软盘的操作。

# 第五步，设置日志文件的输出。
log: bochs.out

# 第六步，开启或关闭某些功能。
# 下面是关闭鼠标，并打开键盘。
mouse: enabled=0
keyboard_mapping: enabled=1,
map=/实际路径/bochs/share/bochs/keymaps/x11-pc-us.map

# 硬盘设置
ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14

# 下面的是增加的 bochs 对 gdb 的支持，这样 gdb 便可以远程连接到此机器的 1234 端口调试了
gdbstub: enabled=1, port=1234, text_base=0, data_base=0, bss_base=0

###################  配置文件结束  #####################
```

好了，现在将上面的配置文件存为 bochsrc.disk 放在 bochs 安装目录下。（bochs 配置文件位置不固定，名字也不要求固定），后缀.disk 是我人为加的，为了表示此配置文件配置的内容是从硬盘启动，这样较明确。

1.5 运行 bochs

终于安装完成了，虽然这过程中有可能会出现各种各样的问题，但还是值得庆祝的，对 Linux 不熟的朋友第一次就搞定了这么个硬货，我理解您此时的喜大普奔之情，哈哈，给大家点赞。顺便说一句，其实平时我们的运维人员为开发环境付出了远比这更多的努力，所有奋战在一线的系统工程师和运维工程师，您们辛苦了。

不过好奇心让我们按捺不住想一探 bochs 容貌，说实在的，我现在就想先运行一下看看，失败又能怎样，无非是报错退出呗，又不会造成实质性的损失。我非常理解大家的心情，虽然现在还差点东西没完成，但作为求知欲强的技术人必须得获得理解和支持，那现在咱们先运行一下 bochs 试试，至少检测下是不是安装正确了，反正不会破坏咱们的电脑，缺什么的时候咱们再创建也不迟。

怕被读者埋怨我太啰嗦，赶紧在 bochs 安装路径下键入 bin/bochs 并赶紧按下了回车，运行效果如图 1-2 所示。

```
[work@localhost bochs]$ bin/bochs
========================================================================
                        Bochs x86 Emulator 2.6.2
                 Built from SVN snapshot on May 26, 2013
                  Compiled on Jul 22 2014 at 16:23:42
========================================================================
------------------------------
Bochs Configuration: Main Menu
------------------------------

This is the Bochs Configuration Interface, where you can describe the
machine that you want to simulate.  Bochs has already searched for a
configuration file (typically called bochsrc.txt) and loaded it if it
could be found.  When you are satisfied with the configuration, go
ahead and start the simulation.

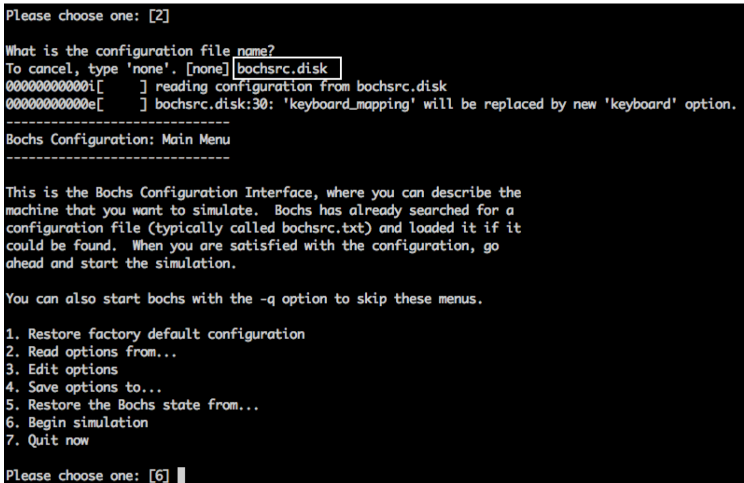
You can also start bochs with the -q option to skip these menus.

1. Restore factory default configuration
2. Read options from...
3. Edit options
4. Save options to...
5. Restore the Bochs state from...
6. Begin simulation
7. Quit now

Please choose one: [2]
```

▲图 1-2　bochs 启动界面 1

看，bochs 界面中给出的提示符默认选项是[2]，Read options from…，这是 bochs 要读取选项的节奏啊，也就是说要读取配置文件，直接按回车键。运行结果如图 1-3 所示。


```
Please choose one: [2]

What is the configuration file name?
To cancel, type 'none'. [none] bochsrc.disk
00000000000i[      ] reading configuration from bochsrc.disk
00000000000e[      ] bochsrc.disk:30: 'keyboard_mapping' will be replaced by new 'keyboard' option.
------------------------------
Bochs Configuration: Main Menu
------------------------------

This is the Bochs Configuration Interface, where you can describe the
machine that you want to simulate.  Bochs has already searched for a
configuration file (typically called bochsrc.txt) and loaded it if it
could be found.  When you are satisfied with the configuration, go
ahead and start the simulation.

You can also start bochs with the -q option to skip these menus.

1. Restore factory default configuration
2. Read options from...
3. Edit options
4. Save options to...
5. Restore the Bochs state from...
6. Begin simulation
7. Quit now

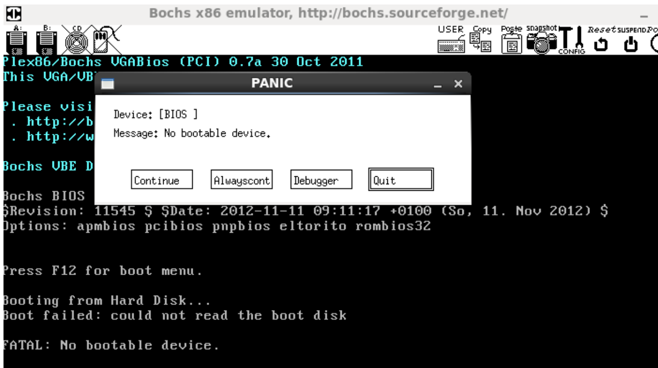
Please choose one: [6]
```

▲图 1-3　bochs 启动界面 2

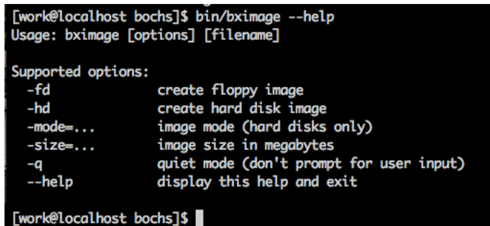
我们键入的是上面长方形框框中的部分：bochsrc.disk。由于我们刚刚把此文件放到了 bochs 的安装路径下，bochs 找到了它并加载成功。紧接着下面给出的默认选项变成了[6]，也就是 Begin simulation 选项，开始模拟 x86 硬件平台。

再多说一句，bochs 如果加载不到配置，它是不会向下运行的，所以在图 1-3 中，白色方框中若不键入配置文件名而直接回车，还是会回到图 1-2 所示的界面，必须给出配置让 bochs 知道您想模拟的硬件是什么才行。

继续回车，马上就有效果了，不过是报错了，如图 1-4 所示。



▲图 1-4　bochs 启动时找不到启动盘


```
[work@localhost bochs]$ bin/bximage --help
Usage: bximage [options] [filename]

Supported options:
  -fd            create floppy image
  -hd            create hard disk image
  -mode=...      image mode (hard disks only)
  -size=...      image size in megabytes
  -q             quiet mode (don't prompt for user input)
  --help         display this help and exit

[work@localhost bochs]$
```

▲图 1-5　bximage 工具

哎哟，不错哦，果然没白测试，报的这是个 PANIC 级别的错误，BIOS 说：“没有启动设备”。

缺什么我们就创建什么，提示没有的这个“bootable device”就是启动盘，现在就创建启动盘吧。

bochs 先生说：“作为一个负责任的模拟器，既然干的就是模拟硬件的工作，那就要把硬件都模拟全了”，所以 bochs 给咱们提供了创建虚拟硬盘的工具 bin/bximage。我们先看下这个命令的帮助，如图 1-5 所示。

-fd 创建软盘。

-hd 创建硬盘。

-mode 创建硬盘的类型，有 flat、sparse、growing 三种。

-size 指创建多大的硬盘，以 MB 为单位。

-q 以静默模式创建，创建过程中不会和用户交互。

按照上面的帮助，那咱们就开工啦，如图 1-6 所示。

```
bin/bximage -hd -mode="flat" -size=60 -q hd60M.img
```

这个命令串中最后一个 hd60M.img 是咱们创建的虚拟硬盘的名称。

```
[work@localhost bochs]$ bin/bximage -hd -mode="flat" -size=60 -q hd60M.img
========================================================================
                                bximage
                  Disk Image Creation Tool for Bochs
          $Id: bximage.c 11315 2012-08-05 18:13:38Z vruppert $
========================================================================

I will create a 'flat' hard disk image with
  cyl=121
  heads=16
  sectors per track=63
  total sectors=121968
  total size=59.55 megabytes

Writing: [] Done.

I wrote 62447616 bytes to hd60M.img.

The following line should appear in your bochsrc:
  ata0-master: type=disk, path="hd60M.img", mode=flat, cylinders=121, heads=16, spt=63
```

▲图 1-6　用 bximage 创建虚拟硬盘

如果大家觉得以上键入命令繁琐，不想用命令行的话，可以直接键入 bin/bximage 回车，后面的提示很清楚，很容易帮助大家创建硬盘。

硬盘创建好了，该如何安装到虚拟机中呢？

看图 1-6 下面的白色方框中的内容，bochs 说：“The following line should appear in your bochsrc：下面的内容应该出现在你的配置文件中”。可见 bochs 的良苦用心，连硬盘的配置都给我们写好了，我们要做的就是复制这些到我们的 bochsrc.disk 中。可见，在 bochs 中有哪些硬件，就是通过配置文件来反映出来的。

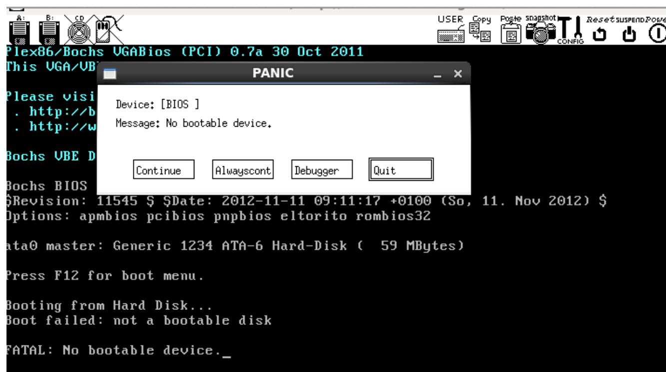
事不宜迟，赶紧更新 bochsrc.disk，找到第 33 行注释部分，将内容添加到 35 行，保存，如图 1-7 所示。

```
33 # 硬盘设置
34 ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
35 ata0-master: type=disk, path="hd60M.img", mode=flat, cylinders=121, heads=16, spt=63
36
```

▲图 1-7　在 bochs 配置文件中增加硬盘

此刻的我已经迫不及待地想看看 bochs 现在的运行情况，不过如果每次启动 bochs 后都要通过 Read options from 选项读取配置文件，这就太麻烦了，其实启动 bochs 的时候，有个更简便的方法，我们用-f 来指定其配置文件便可。

bin/bochs –f bochsrc.disk 回车，观察效果，如图 1-8 所示。



▲图 1-8　效果图

看上去和图 1-4 报错一样，都是提示没有启动盘。这是怎么回事呢？仔细看过之后，发现这里的报错和图 1-4 还是有些不同的，虽然结果是一样的错误，但原因是不同的。图 1-4 中的报错原因是 boot failed: could not read the boot disk，这是无法读取启动盘。而现在这里的报错是 boot failed: not a bootable disk，这不是一个启动盘。这两个原因明显不是一码事，就像某件衣服穿着不合适一样，原因是一个人是太胖了，另一个人是太瘦了。

不要灰心，这正是我们在下一章要讲的内容，什么才算启动盘，真正的启动盘上有什么。本章到此结束，下章我们再见。

第 2 章　编写 MBR 主引导记录，让我们开始掌权

2.1 计算机的启动过程

不知道大家对“载入内存”这 4 个字的理解是怎样的。以下这两点是我曾经的疑问：第一，为什么程序要载入内存。第二，什么是载入内存。

先回答第一个。

CPU 的硬件电路被设计成只能运行处于内存中的程序，这是硬件基因的问题，这样做的原因，首先肯定是内存比较快，且容量大。

其次，操作系统可以存储在软盘上，也可以存储在硬盘上，甚至 U 盘，当然还有很多存储介质都可以。但由于各个硬件特性不同，操作系统要分别考虑每种硬件的特性才行。所以，都在内存中运行程序，操作系统和硬件设计都省事了，这可能也是为了方式的统一吧，否则总不能出现某种存储介质后，操作系统和硬件就要付出额外努力去支持。当然，具体原因只有硬件工程师才知道，咱们在此先打住，继续咱们的内容。

马上回答第二个。

老听说“程序载入内存”，我不知道有多少同学对这个词仅仅是感性认识。

我隐约觉得很多同学都会将“载入内存”和“程序执行”画等号。所谓的载入内存，大概上分两部分。

（1）程序被加载器（软件或硬件）加载到内存某个区域。

（2）CPU 的 cs：ip 寄存器被指向这个程序的起始地址。

操作系统在加载程序时，是需要某个加载器来将用户程序存储到内存中的。其实“加载器”这只是人为起的名字，突显了其功能，并不是多么神秘的东西，本质上它就是一堆函数组成的模块，不要因为未知的东西而感到畏惧。

从按下主机上的 power 键后，第一个运行的软件是 BIOS。于是产生了三个问题。

（1）它是由谁加载的。

（2）它被加载到哪里。

（3）它的 cs：ip 是谁来更改的。

2.2 软件接力第一棒，BIOS

BIOS 全称叫 Base Input & Output System，即基本输入输出系统。

人们给任何事物起名字，肯定都不是乱起的，必然是根据该事物的特点，通过总结，精练出一些文字来标识此事物，这个便是对一般事物取名的方法。通过名字，就能够反应出该事物的特性。最符合特性的名字就是昵称和外号了，比如抽油机是用来开采石油的一种机器，因为其工作时，就像“磕头”一样，所以大家给其起了更形象的名字——“磕头机”。

回到 BIOS 上，输入输出我理解，命名中加上系统二字也明白，可为什么还要用“基本”来修饰呢？不知道您是不是和我一样喜欢咬文嚼字，我们必须得把它搞清楚。

2.2.1　实模式下的 1MB 内存布局

先来点背景知识，很久很久以前：

Intel 8086 有 20 条地址线，故其可以访问 1MB 的内存空间，即 2 的 20 次方=1048576=1MB，地址范

围若按十六进制来表示，是 0x00000 到 0xFFFFF。不知道硬件工程师当时设计的初衷是什么，总之人家有自己的理由，这 1MB 的内存空间被分成多个部分。

为了让大家先有个印象，免得太抽象不容易理解，先把实模式下 1MB 内存给大家梳理一下，很辛苦的，各位看官要仔细看哈，所以感兴趣或有强迫症的同学一定要背下来（玩笑），见表 2-1。

表 2-1　　实模式下的内存布局

起始	结　束	大　小	用　途
FFFF0	FFFFF	16B	BIOS 入口地址，此地址也属于 BIOS 代码，同样属于顶部的 640KB 字节。只是为了强调其入口地址才单独贴出来。此处 16 字节的内容是跳转指令 jmp f000：e05b
F0000	FFFEF	64KB-16B	系统 BIOS 范围是 F0000～FFFFF 共 640KB，为说明入口地址，将最上面的 16 字节从此处去掉了，所以此处终止地址是 0XFFFEF
C8000	EFFFF	160KB	映射硬件适配器的 ROM 或内存映射式 I/O
C0000	C7FFF	32KB	显示适配器 BIOS
B8000	BFFFF	32KB	用于文本模式显示适配器
B0000	B7FFF	32KB	用于黑白显示适配器
A0000	AFFFF	64KB	用于彩色显示适配器
9FC00	9FFFF	1KB	EBDA（Extended BIOS Data Area）扩展 BIOS 数据区
7E00	9FBFF	622080B 约 608KB	可用区域
7C00	7DFF	512B	MBR 被 BIOS 加载到此处，共 512 字节
500	7BFF	30464B 约 30KB	可用区域
400	4FF	256B	BIOS Data Area（BIOS 数据区）
000	3FF	1KB	Interrupt Vector Table（中断向量表）

先从低地址看，地址 0～0x9FFFF 处是 DRAM（Dynamic Random Access Memory），即动态随机访问内存，我们所装的物理内存就是 DRAM，如 DDR、DDR2 等。又要开始咬文嚼字了，动态是什么意思？动态指此种存储介质由于本身电气元件的性质，需要定期地刷新。内存中的每一位都是由电容和晶体管来组成的，您想，单条内存现在都到 4GB，内存条的体积大小您也清楚，那么小的面积得集成多少电容才能够拼凑出 4GB 的内存容量，不包括相关电路元件，也得是 4GB×8 个电容了。如此小的电容，其缺点也是明显的，漏电很快，所以漏电了就要及时把电补充上去，这样数据才不至于丢失。这个补充电的过程就称为刷新。其实不仅是电容需要刷新，就连电信号也是一样的，不知道您注意了没有，我们平时使用的网线，也是需要在每隔一定长度距离时接个中继放大器，这个就是来放大电信号的，因为物理链路一长，信号衰减就特别严重，只好通过这种“打气”的方式来保持稳定了。终于把动态这一词搞定了，不过我们最终要搞定的词是 BIOS 中的“基本”，所以咱们还得接着看。

见表 2-1，内存地址 0～0x9FFFF 的空间范围是 640KB，这片地址对应到了 DRAM，也就是插在主板上的内存条。有没有人开始小声嘀咕了：为什么是对应到了 DRAM，难道不是直接访问到我的物理内存 DRAM 吗？难道我的内存条不是全部的内存？还可以访问到别处吗？如果您有这样的疑问，我除了回答是啊是啊之外，还是很欣慰的，终于有人和我之前想的一样了。

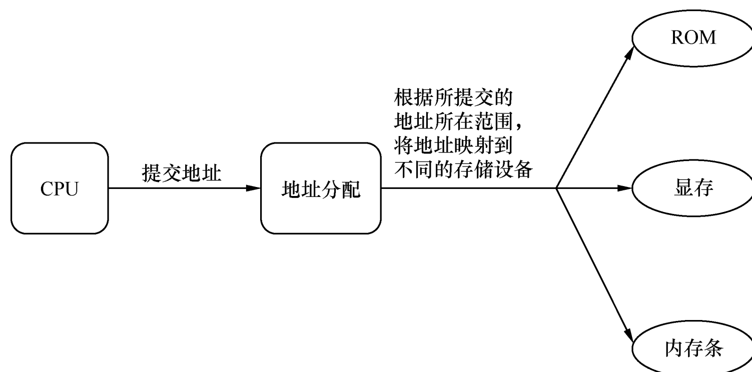
一会再解释这个，否则咱们离“基本”越来越远了。表 2-1，看顶部的 0xF0000～0xFFFFF，这 64KB 的内存是 ROM。这里面存的就是 BIOS 的代码。BIOS 的主要工作是检测、初始化硬件，怎么初始化的？硬件自己提供了一些初始化的功能调用，BIOS 直接调用就好了。BIOS 还做了一件伟大的事情，建立了中断向量表，这样就可以通过“int 中断号”来实现相关的硬件调用，当然 BIOS 建立的这些功能就是对硬件的 IO 操作，也就是输入输出，但由于就 64KB 大小的空间，不可能把所有硬件的 IO 操作实现得面面俱到，而且也没必要实现那么多，毕竟是在实模式之下，对硬件支持得再丰富也白搭，精彩的世界是在进入保护模式以后才开始，所以挑一些重要的、保证计算机能运行的那些硬件的基本 IO 操作，就行了。这就是 BIOS 称为基本输入输出系统的原因。

现在开始解释另一个问题，在 CPU 眼里，为什么我们插在主板上的物理内存不是它眼里“全部的内存”。

地址总线宽度决定了可以访问的内存空间大小，如 16 位机的地址总线为 20 位，其地址范围是 1MB，

32 位地址总线宽度是 32 位，其地址范围是 4GB。但以上的地址范围是指地址总线可以触及到的边界，是指计算机在寻址上可以到达的疆域。可是人家并没有说要寻哪里，就拿 16 位机来说，并没有说这 1MB 的寻址范围必须得是物理内存（内存条），难道人家 20 位的地址总线就认得这一亩三分地？完全不是。

归根结底的原因是这样的：在计算机中，并不是只有咱们插在主板上的内存条需要通过地址总线访问，还有一些外设同样是需要通过地址总线来访问的，这类设备还很多呢。若把全部的地址总线都指向物理内存，那其他设备该如何访问呢？由于这个原因，只好在地址总线上提前预留出来一些地址空间给这些外设用，这片连续的地址给显存，这片连续的地址给硬盘控制器等。留够了以后，地址总线上其余的可用地址再指向 DRAM，也就是指插在主板上的内存条、我们眼中的物理内存。示意如图 2-1 所示。



▲图 2-1　地址映射

物理内存多大都没用，主要是看地线总线的宽度。还要看地址总线的设计，是不是全部用于访问 DRAM。所以说，地址总线是决定我们访问哪里、访问什么，以及访问范围的关键。我们平时用的机器一般是 32 位，上面的内存条并不是全部都用到了，按理说内存条大小超过 4GB 就没意义了，超过了地址总线的势力就是浪费。不过通过前面的介绍，即使内存条大小没有超过地址总线的范围，也不会全都能被访问到，毕竟要预留一些地址用来访问其他外设，所以最终还得看地址总线把地址指向哪块内存了。这就是安装了 4GB 内存，电脑中只显示 3.8GB 左右的原因。

总之，表示地址的那串数字是地址总线的输入，相当于其参数，和内存条没关系。CPU 能够访问一个地址，这是地址总线给做的映射，相当于给该地址分配了一个存储单元，而该存储单元要么落在某个 rom 中，要么落到了某个外设的内存中，要么落到了物理内存条上。可以想像成，CPU 给地址总线提交一个数字，在地址总线看来，这串数字就是地址。地址分配电路根据此地址的范围，决定在哪个存储介质中分配一个存储单元，最后将此地址与此存储单元对应起来。当然事实上未必是这样，我刚才说了，可以想像成这样。我们学习新的知识，很多时候都是建立在原有的知识上，用原有的知识帮助学习新的知识，就像第一次听说电动车的时候，我们潜意识里是用车和蓄电池的概念在联想电动车的形象。如果要学的是一种全新的知识，并且无从用旧的知识来辅助学习时，试图靠想像力是非常有效的。对于知识的掌握，这并没有什么标准，每个人对知识的理解都是不同的，即使两个人都考了满分，其思考过程也是不同的。所以，对于一个新知识的掌握，本质上是给了一个能够说服自己的理由，能够自圆其说，这就够了。

2.2.2　BIOS 是如何苏醒的

BIOS 其实一直睡在某个地方，直到被唤醒……

前面热火朝天地说了 BIOS 的功能和内存布局，似乎还没说到正题上，BIOS 是如何启动的呢？因为 BIOS 是计算机上第一个运行的软件，所以它不可能自己加载自己，由此可以知道，它是由硬件加载的。那这个硬件是谁呢？其实前面已经提到过了，相当于是只读存储器 ROM，因为它一直就睡在那里不动。

大家知道，只读存储器中的内容是不可擦除的，也就是它不像动态随机访问存储器 DRAM 那样，掉电后，里面的数据就会丢失。这种存储介质是用来存储一成不变的数据的，当数据写进去后，便与日月同

辉，庭前坐看花开花落，不朽于天地万物之间，哈哈，有点夸张了。

BIOS 代码所做的工作也是一成不变的，而且在正常情况下，其本身是不需要修改的，平时听说的那些主板坏了要刷 BIOS 的情况属于例外。于是 BIOS 顺理成章地便被写进此 ROM。ROM 也是块内存，内存就需要被访问。此 ROM 被映射在低端 1MB 内存的顶部，即地址 0xF0000～0xFFFFF 处，可以参考表 2-1 顶部的 BIOS 部分。只要访问此处的地址便是访问了 BIOS，这个映射是由硬件完成的。

BIOS 本身是个程序，程序要执行，就要有个入口地址才行，此入口地址便是 0xFFFF0。

最重要的一点来了，知道了 BIOS 在哪里后，CPU 如何去执行它，即 CPU 中的 cs：ip 值是如何组合成 0xFFFF0 的。

如果大家不了解内存的分段访问机制，可以参考第 0 章，里面有讲解 CPU 为什么分段方式内存。说正事，CPU 访问内存是用段地址+偏移地址来实现的，由于在实模式之下，段地址需要乘以 16 后才能与偏移地址相加，求出的和便是物理地址，CPU 便拿此地址直接用了。这个“段基址：段内偏移地址”的组合是 0xffff：0 吗？或者是 0xF000：0xFFF0?或者是更奇葩一点的组合：0xFEEE：0x1110？或者您想出的组合比我的还奇葩，好啦，不折磨大家了，还是说正事要紧。既然作为第一个运行的程序都没开始执行，自然就没办法用软件搞定这件事了，还是得靠硬件支持才行。

在开机的一瞬间，也就是接电的一瞬间，CPU 的 cs：ip 寄存器被强制初始化为 0xF000：0xFFF0。由于开机的时候处于实模式，再重复一遍加深印象，在实模式下的段基址要乘以 16，也就是左移 4 位，于是 0xF000：0xFFF0 的等效地址将是 0xFFFF0。上面说过了，此地址便是 BIOS 的入口地址。

当我给出这个地址后，不知道大家意识到什么没有。BIOS 是在实模式下运行的，而实模式只能访问 1MB 空间（20 位地址线，2 的 20 次方是 1MB）。而地址 0xFFFF0 距 1MB 只有 16 个字节了（见表 2-1 除标题外的第一行），这么小的空间够干吗？BIOS 又要检测硬件，做各种初始化工作，还要建立中断向量表……16 字节的机器指令肯定干不了这么多事。也许有的同学会问，超过寄存器宽度会怎么样呢？比如 0xFFFF0+16，这样就溢出了，由于实模式下的寄存器宽度是 16 位，0xFFFF0+16 已经超过了其最大值 0xFFFFF。溢出的部分就会回卷到 0，又会重新开始，即 0xFFFF0+16 等于 0，0xFFFF0+17 等于 1。

既然此处只有 16 字节的空间了，这只能说明 BIOS 真正的代码不在这，那此处的代码只能是个跳转指令才能解释得通了。好，既然心里有了推断，那咱们就来证明这个推断正确与否。

图 2-2 是我在 bochs 中抓的图，下面给大家分析一下这图中的信息都代表什么。

```
Please choose one: [6]
00000000000i[     ] installing x module as the Bochs GUI
00000000000i[     ] using log file bochs.out
Next at t=0
(0) [0x0000fffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ; ea5be000
f0                   cs : ip                下一条待执行的指令是jmp 0xf000:0xe05b
                                            跳转到物理地址0xfe05b
<bochs:1> sreg
es:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
cs:0xf000, dh=0xff0093ff, dl=0x0000ffff, valid=7
        Data segment, base=0xffff0000, limit=0x0000ffff, Read/Write, Accessed
ss:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ds:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
fs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
gs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
        Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ldtr:0x0000, dh=0x00008200, dl=0x0000ffff, valid=1
tr:0x0000, dh=0x00008b00, dl=0x0000ffff, valid=1
gdtr:base=0x00000000, limit=0xffff
idtr:base=0x00000000, limit=0xffff
<bochs:2>
```

▲图 2-2　bochs 开机界面

首先得承认，这张图有点超前了，这是在有了 MBR 后才能抓到的，否则会提示 boot failed： not a bootable disk，而我们还没有 MBR，还没有写主引导记录。先不管这张图是怎么来的啦，反正大家立即就能够在自己的虚拟机里看到这张图了。大家先注意框框中的内容。一共有 3 个，最上面左边第 1 个标有 cs：ip 的那个框，cs 寄存器的值是 0xf000，ip 寄存器的值是 0xfff0，也就是段基址 0xf000，段内偏移地址 0xfff0，

这个组合出来的地址便是 0xffff0，这是处理器下一条待执行指令的地址。这与上面所说的 BIOS 入口地址是吻合的。另外，因为 cs 和 ip 寄存器中存储的是下一条要执行的指令，目前还没有执行，也就是说，当前还没有执行 BIOS，这是机器刚开机的那一刻。这一刻还是值得庆祝的，因为即使是计算机行业的同学都很少看到这一刻，何况我们让这一刻停了下来，成为永恒。

按理说，既然让 CPU 去执行 0xFFFF0 处的内容（目前还不知道其是指令，还是数据），此内容应该是指令才行，否则这地址处的内容若是数据，而不是指令，CPU 硬是把它当成指令来译码的话，一定会弄巧成拙铸成大错。现在咱们又有了新的推断，物理地址 0xFFFF0 处应该是指令，继续探索。

继续看第二个框框，里面有条指令 jmp far f000：e05b，这是条跳转指令，也就是证明了在内存物理地址 0xFFFF0 处的内容是一条跳转指令，我们的判断是正确的。那 CPU 的执行流是跳到哪里了呢？段基址 0xf000 左移 4 位+0xe05b，即跳向了 0xfe05b 处，这是 BIOS 代码真正开始的地方。

第三个框框 cs：f000，其意义是 cs 寄存器的值是 f000，与我们刚刚所说的加电时强制将 cs 置为 f000 是吻合的，正确。

接下来 BIOS 便马不停蹄地检测内存、显卡等外设信息，当检测通过，并初始化好硬件后，开始在内存中 0x000～0x3FF 处建立数据结构，中断向量表 IVT 并填写中断例程。

好了，终于到了接力的时刻，这是这场接力赛的第一棒，它将交给谁呢？咱们下回再说。

2.2.3　为什么是 0x7c00

计算机执行到这份上，BIOS 也即将完成自己的历史使命了，完成之后，它又将睡去。想到这里，心中不免一丝忧伤，甚至有些许挽留它的想法。可是，这就是它的命，它生来被设计成这样，在它短暂的一生中已经为后人创造了足够的精彩。何况，在下一次开机时，BIOS 还会重复这段轮回，它并没有消失。好了，让伤感停止，让梦想前行。

先说重点，BIOS 最后一项工作校验启动盘中位于 0 盘 0 道 1 扇区的内容。

在此插播一段小告示：在计算机中是习惯以 0 作为起始索引的，因为人们已经习惯了偏移量的概念，无论是机器眼里和程序员眼里，用“相对”的概念，即偏移量来表示位置显得很直观，所以很多指令中的操作数都是用偏移量表示的。0 盘 0 道 1 扇区本质上就相当于 0 盘 0 道 0 扇区。为什么称为 1 呢，因为硬盘扇区的表示法有两种，我们描述 0 盘 0 道 1 扇区用的便是其中的一种：CHS 方法，即柱面 Cylinder 磁头 Header 扇区 Sector（另外一种是 LBA 方式，暂不关心），“0 盘”说的是 0 磁头，因为一张盘是有上下两个盘面的，一个盘面上对应一个磁头，所以用磁头 Header 来表示盘面。“0 道”是指 0 柱面，柱面 Cylinder 指的是所有盘面上、编号相同的磁道的集合，形象一点描述就是把很多环叠摞在一起的样子，组合在一起之后是一个立体的管状。“1 扇区”才是我们要解释的部分，将磁道等距划分成一段段的小区间，由于磁道是圆形的，确切地说是圆环，这些被划分出来的小区间便是扇形，所以称为扇区。好了，背景交待完了，重点来了，在 CHS 方式中扇区的编号是从 1 开始的，不是 0，不是 0，原谅我说了两次，良苦用心你懂的，所以 0 盘 0 道 1 扇区其实就相当于 0 盘 0 道 0 扇区，它就是磁盘上最开始的那个扇区。而 LBA 方式中，扇区编号是从 0 开始的。关于硬盘的知识我会在以后章节专门来讲，这里我若没表达清楚，大家先不要着急，只要知道 MBR 所在的位置是磁盘上最开始的那个扇区就行了。

继续说，如果此扇区末尾的两个字节分别是魔数 0x55 和 0xaa，BIOS 便认为此扇区中确实存在可执行的程序（在此先剧透一下，此程序便是久闻大名的主引导记录 MBR），便加载到物理地址 0x7c00，随后跳转到此地址，继续执行。

这里有个小细节，BIOS 跳转到 0x7c00 是用 jmp 0：0x7c00 实现的，这是 jmp 指令的直接绝对远转移用法，段寄存器 cs 会被替换，这里的段基址是 0，即 cs 由之前的 0xf000 变成了 0。

如果此扇区的最后 2 个不是 0x55 和 0xaa，即使里面有可执行代码也无济于事了，BIOS 不认，它也许还认为此扇区是没格干净呢。

不过，这就又抛出两个问题。

（1）为什么是 0 盘 0 道 1 扇区的内容？

（2）为什么是物理地址 0x7c00，而不是个好记或好看的其他地址？

先回答第 1 个，我想这个问题不用官方解释了，因为官方确实没什么好说的，不过他们出于尊重客户，还是会像我一样说出类似下面的话。

我就个人观点给大家一个理由，未经核实，仅是自己一面之词，请大家提高警惕，小心谨慎^—^。

在计算机中处处充满了协议、约定，所以，将 0 盘 0 道 1 扇区作为 mbr 的栖身之地，我完全可以理解为规定。我们反证一下，如果不存在这个“规定”，会发生什么。当然，此扇区最初是给 BIOS 使用的，咱们设想一下 BIOS 的工作将变成怎样。

主引导记 mbr 是段程序，无论位于软盘、硬盘或者其他介质，总该有个地方保存它。Ok，现在不告诉 BIOS 它存储在哪个位置了。BIOS 只好将所有检测到的存储设备上的每一个存储单位都翻一遍，挨个对比，如果发现该存储单位最后的两个字节是 0x55 和 0xaa，就认为它是 mbr。这就好比查字典一样，不用偏旁部首和拼音检索的方法，只能一页一页翻了。

几经花开花落，找到 mbr 的那一刻，BIOS 满脸疲惫地说：“你是我找了好久好久的那个人”。mbr 抬起经不起岁月等待的脸：“难得你还认得我，我等你等到花儿都谢了”。其实 BIOS 的心声是：“看我手忙脚乱的样子，你们这是要闹哪样啊。就那么 512 字节的内容，害我找遍全世界，我们是在跑接力赛啊，下一棒的选手我都不知道在哪里……以后让它站在固定的位置等我！”

由于 0 盘 0 道 1 扇区是磁盘的第一个扇区，mbr 选择了离 BIOS 最近的位置站好了，从此以后再也不担心被 BIOS 骂了。

计算机中处处有固定写死的东西，还用举个例子吗？不用了吧？因为任何一个魔数都是啊，有请下一个魔数 0x7c00 登场。

至于 0x7c00，很久之前，比我好奇心大的人查遍了 Intel 开发手册都没找到相关的说明。要想知道事情的来龙去脉，还是要从个人计算机的初始说起，同样是很久很久以前……

1981 年 8 月，IBM 公司生产了世界上第一台个人计算机 PC 5150，所以它就是现代 x86 个人计算机兼容机的祖先。说到有关历史的东西，不给来点真相就感觉气场不足，图 2-3 所示便是 IBM PC 5150，有没有感受到计算机文化底蕴呢？



▲图 2-3 IBM PC 5150

既然 Intel 开发手册中没有相关说明，那咱们就朝其他方向找答案，换句话说，既然不是 CPU 的硬性规定，那很可能就是代码中写死的。为了搞清楚 0x7c00 是哪里来的，咱们先探索下“IBM PC 5150”的 BIOS 的秘密。请先深深呼吸一大口气，“0x7C00”最早出现在 IBM 公司出产的个人电脑 PC5150 的 ROM BIOS 的 INT19H 中断处理程序中，说了这么多定语，感觉气都喘不上来了。

通电开机之后，BIOS 处理程序开始自检，随后，调用 BIOS 中断 0x19h，即 call int 19h。在此中断处理函数中，BIOS 要检测这台计算机有多少硬盘或软盘，如果检测到了任何可用的磁盘，BIOS 就把它的第一个扇区加载到 0x7c00。

现在应该搞清楚了为什么在 x86 手册里找不到它的说明了，它是属于 BIOS 中的规范。似乎这下好办了，既然是 BIOS 中的规范，那肯定是 IBM PC 5150 BIOS 开发团队规定的这个数。

个人计算机肯定要运行操作系统，在这台计算机上，运行的操作系统是 DOS 1.0，不清楚此系统要求的最小内存是 16KB，还是 32KB，反正 PC 5150 BIOS 研发工程师就假定其是 32KB 的，所以此版本 BIOS 是按最小内存 32KB 研发的。

MBR 不是随便放在哪里都行的，首先不能覆盖已有的数据，其次，不能过早地被其他数据覆盖。不覆盖已有数据，这个好理解。说一下后面这个“其次”。通常，MBR 的任务是加载某个程序（这个程序一般是内核加载器，很少有直接加载内核的）到指定位置，并将控制权交给它。所谓的交控制权就是 jmp 过去而已。之后 MBR 就没用了，被覆盖也没关系。我说的过早被覆盖，是指不能让 mbr 破坏自己，比如被加

载的程序，如内核加载器，其放置的内存位置若是 MBR 自己所在的范围，这不就是破坏自己了吗，这就是我所说的“过早”了，怎么也得等 MBR 执行完才行。

重现一下当时的内存使用情况。

8086CPU 要求物理地址 0x0～0x3FF 存放中断向量表，所以此处不能动了，再选新的地方看看。

按 DOS 1.0 要求的最小内存 32KB 来说，MBR 希望给人家尽可能多的预留空间，这样也是保全自己的作法，免得过早被覆盖。所以 MBR 只能放在 32KB 的末尾。

MBR 本身也是程序，是程序就要用到栈，栈也是在内存中的，MBR 虽然本身只有 512 字节，但还要为其所用的栈分配点空间，所以其实际所用的内存空间要大于 512 字节，估计 1KB 内存够用了。

结合以上三点，选择 32KB 中的最后 1KB 最为合适，那此地址是多少呢？32KB 换算为十六进制为 0x8000，减去 1KB(0x400)的话，等于 0x7c00。这就是倍受质疑的 0x7c00 的由来，这下清楚了。

可见，加载 MBR 的位置取决于操作系统本身所占内存大小和内存布局。

我想大家现在都心痒痒了吧，说了这么久，CPU 中运行的都是 BIOS 的代码，连自己一句代码都没跑起来呢。事不宜迟，马上写一个 MBR，先让它跑起来再说。

2.3 让 MBR 先飞一会儿

虽说主引导记录 mbr 是咱们能够掌控的第一个程序，但这并不是让我们为之激动的理由。我们平时所写的程序都要依赖于操作系统，而我们即将实现的这个程序是独立于操作系统的，能够直接在裸机上运行，这才是让我们激动的理由，对咱们来说这无疑是历史性的一刻。还记得当初我的 MBR 跑起来时，那可真是发自内心的高兴呀。

好了，不再抒情了，说正事要紧。MBR 的大小必须是 512 字节，这是为了保证 0x55 和 0xaa 这两个魔数恰好出现在该扇区的最后两个字节处，即第 510 字节处和第 511 字节处，这是按起始偏移为 0 算起的。由于我们的 bochs 模拟的是 x86 平台，所以是小端字节序，故其最后两个字节内容是 0xaa55，写到一起后似乎有点不认识了，不要怕，拆开就是 0x55 和 0xaa。

2.3.1 神奇好用的$和$$，令人迷惑的 section

$和$$是编译器 NASM 预留的关键字，用来表示当前行和本 section 的地址，起到了标号的作用，它是 NASM 提供的，并不是 CPU 原生支持的，相当于伪指令一样，对 CPU 来说是假的。

指令本来没有真伪之别，就像酒一样，因为有了假酒，所以才有了真酒之说。伪指令是相对于 CPU 可识别的指令来说的，它（伪指令）只是编译器定义的，CPU 中并不存在这个指令，愣让 CPU 执行这些伪指令，CPU 会抛出“UD（未定义的操作码）”异常。伪指令是编译器为了开发人员写代码方便而提供的一些符号，这些符号在编译时，会由编译器转换成 CPU 可识别的东西，如指令或地址等。

汇编语言中的标号是程序员“显式地”写在明处的，如：

```
……
code_start：
   mov ax， 0
……
```

code_start 这个标号被 nasm 认为是一个地址，此地址便是“mov ax，0”这条指令所在的地址，即其指令机器码存放的内存位置是 code_start。code_start 只是个标记，CPU 并不认识，和伪指令类似，它是假的，CPU 不认。所以 nasm 会用为其安排的地址来替换标号 code_start，到了 CPU 手中，已经被替换为有意义的数字形式的地址了。

$属于“隐式地”藏在本行代码前的标号，也就是编译器给当前行安排的地址，看不到却又无处不在，$在每行都有。或者这种说法并不是很正确，只有“显示地”用了$的地方，nasm 编译器才会将此行的地址公布出来。如果上面的例子改为：

```
……
```

```
code_start:
   jmp $
……
```

这就和 jmp code_start 是等效的。$和 code_start 是同一个值。

$$指代本 section 的起始地址，此地址同样是编译器给安排的。

对于$和$$的意义，我强调过了，是编译器给安排的地址，默认情况下，它们的值是相对于本文件开头的偏移量。至于实际安排的是多少，还要看程序员同学是否在 section 中添加了 vstart。这个关键字可以影响编译器安排地址的行为，如果该 section 用了 vstart=xxxx 修饰，$$的值则是此 section 的虚拟起始地址 xxxx。$的值是以 xxxx 为起始地址的顺延。如果用了 vstart 关键字，想获得本 section 在文件中的真实偏移量（真实地址）该怎么做？nasm 编译器提供了这个方法。

section.节名.start。

如果没有定义 section，nasm 默认全部代码同为一个 section，起始地址为 0。

稍带说一下 section。很多东西从名字上就能理解它的功能，毕竟名字不是乱起的。section 也称为节、段，故名思义，是程序中的一小块，形象一点地说，就是用 section 这个关键字在程序中圈出一块地，并向编译器宣称，这块地我要做些规划，至于我用来干什么您就不用操心了，编译时请您合理安排。

为什么说合理安排呢，因为 section 是伪指令，是 nasm 提供的，具体解释权还是人家 nasm 说了算。比如以下代码：

```
section data
    var dd 0
section code
    jmp start
… ...
```

编译器一看这两个 section，data 中定义的是变量，code 中是代码，于是把这两个 section 的内容分别归入最终的数据段和代码段。

有时候 nasm 并不会完全听您的，如改为下面的例子：

```
section data_a
   var dd 0
section code
   jmp start
section data_b
   var dd 1
   ……
```

虽然人为定义了三个 section，但 nasm 发现 data_a 和 data_b 这两个 section 完全能够合并到一起，于是在编译阶段会被“合理”地安排到一起。

在第 0 章中有说明 section 和 segment 的区别。section 是伪指令，CPU 运行程序是不需要这个东西的，这个只是用来给程序员规划程序用的，有了 section，就可以将自己的代码分成一段一段的，当然这只是在逻辑上的段，实际上编译出来的程序还是完整的一体。逻辑上划分成段的好处是方便开发人员梳理代码，方便管理。想像一下，把一大片农田按亩来划分成一个个的小段，一眼望去，是不是显得井然有序呢？单是简短的几行汇编代码是无法体现出这一优势的，就像如果农田本来就不大，还要划分成多个段，那自然是得不偿失的。当代码量上去的时候，会发现如果不在逻辑上将其拆分成几块，对一锅粥似的代码进行维护，代价还是很大的，可能一会儿脑子也像一锅粥了呢。

划分成 section 后，编译器便根据您的意图，将这些 section 中的内容安排位置，它被安排到哪里咱们是不需要关心的，咱们也不必管，因为程序内部的关联是通过地址实现的。想想看，无非是 section 被安排到 A 位置，其他用到此 section 中内容的相关指令，其操作数为 A 地址，若 section 被安排到 B 位置，操作数便是 B 地址，这些都是编译器安排的，它会帮您圆上的。

关于 section 地址更详细的说明，大家可以参照第 3 章，这里只是抛砖引玉。

总之，section 是给开发人员逻辑上规划代码用的，只起到思路清晰的作用，最终还是在编译阶段由 nasm 在物理上的规划说了算。

2.3.2 NASM 简单用法

在咱们的实际工程中只用到了 nasm 的一些简单功能，所以不必担心连操作系统的一句代码都没写呢，却先要为学习其他的东西而付出额外的精力。

```
nasm -f <format><filename> [-o <output>]
```

以上是 nasm 的基本用法，对咱们来说，够用了。注意我说的是“基本”，还有好多其他参数呢，不过咱们用不着。甚至，大多数时候连-f 都不用呢。

-o 就是指定输出可执行文件的名称。

查看一下 nasm 的帮助，ok，执行 man nasm 回车，输出的信息太多了，我们只看-f 的说明就行了。

```
-f format
     Specifies the output file format.
To see a list of valid output formats， use the -hf option.
```

瞧，人家说啦，-f 是用来指定输出文件的格式。要想知道有多少种有效的输出格式，用-hf 选项。那咱们还是用 nasm –hf 来查看一下吧，见表 2-2。

表 2-2 nasm 编译输出的格式

格 式	描 述
bin	flat-form binary files (e.g. DOS .COM, .SYS），此项为默认
ith	Intel hex
srec	Motorola S-records
aout	Linux a.out object files
aoutb	NetBSD/FreeBSD a.out object files
coff	COFF (i386) object files (e.g. DJGPP for DOS)
elf32	ELF32 (i386) object files (e.g. Linux)
elf64	ELF64 (x86_64) object files (e.g. Linux)
elfx32	ELFX32 (x86_64) object files (e.g. Linux)
as86	Linux as86 (bin86 version 0.3) object files
obj	MS-DOS 16-bit/32-bit OMF object files
win32	Microsoft Win32 (i386) object files
win64	Microsoft Win64 (x86-64) object files
rdf	Relocatable Dynamic Object File Format v2.0
ieee	IEEE-695 (LADsoft variant) object file format
macho32	NeXTstep/OpenStep/Rhapsody/Darwin/MacOS X (i386) object files
macho64	NeXTstep/OpenStep/Rhapsody/Darwin/MacOS X (x86_64) object files
dbg	Trace of all info passed to output stage
elf	ELF (short name for ELF32)
macho	MACHO (short name for MACHO32)
win	WIN (short name for WIN32)

一共列出了 21 个，不过大部分格式和咱们关系不大，咱们只关注 bin 和 elf 格式就好啦。

既然 bin 是默认输出格式，也就是不用-f bin 来明确指定了，所以以后咱们只在输出 elf 格式时才用-f 指定。

bin 是指纯二进制。二进制就二进制吧，还有不纯的？就像前面的拿酒举例一样，本来没有真酒之说，由于有了假酒的出现，才有了真的说法。纯二进制就是不掺杂其他的东西，直接给 CPU 后就能用，也就是可执行文件中什么样，内存中就什么样。我们平时所说的 elf 或 pe 格式的二进制可执行文件，那里面有好多和指令无关的东西，里面掺杂了程序的内存布局、位置等信息，这是给操作系统中的程序加载器用的，是属于操作系统规划的范畴了。

2.3.3 请下一位选手 MBR 同学做准备

有点不好意思了，说了好久，才说到实质性的东西，好了，赶紧说正题。

代码 2-1 （c2/a/boot/mbr.S）

```
 1 ;主引导程序
 2 ;------------------------------------------------------------
 3 SECTION MBR vstart=0x7c00
 4    mov ax,cs
 5    mov ds,ax
 6    mov es,ax
 7    mov ss,ax
 8    mov fs,ax
 9    mov sp,0x7c00
10
11 ; 清屏利用 0x06 号功能,上卷全部行,则可清屏。
12 ; -----------------------------------------------------------
13 ;INT 0x10   功能号:0x06    功能描述:上卷窗口
14 ;------------------------------------------------------
15 ;输入:
16 ;AH 功能号= 0x06
17 ;AL = 上卷的行数(如果为 0,表示全部)
18 ;BH = 上卷行属性
19 ;(CL,CH) = 窗口左上角的(X,Y)位置
20 ;(DL,DH) = 窗口右下角的(X,Y)位置
21 ;无返回值:
22    mov    ax, 0x600
23    mov    bx, 0x700
24    mov    cx, 0              ; 左上角: (0, 0)
25    mov    dx, 0x184f         ; 右下角: (80,25),
26                              ; VGA 文本模式中,一行只能容纳 80 个字符,共 25 行。
27                              ; 下标从 0 开始,所以 0x18=24,0x4f=79
28    int    0x10               ; int 0x10
29
30 ;;;;;;;;;    下面这三行代码获取光标位置    ;;;;;;;;;
31 ;.get_cursor 获取当前光标位置,在光标位置处打印字符。
32    mov ah, 3                  ; 输入: 3 号子功能是获取光标位置,需要存入 ah 寄存器
33    mov bh, 0                  ; bh 寄存器存储的是待获取光标的页号
34
35    int 0x10                   ; 输出: ch=光标开始行,cl=光标结束行
36                               ; dh=光标所在行号,dl=光标所在列号
37
38 ;;;;;;;;;    获取光标位置结束    ;;;;;;;;;;;;;;;;
39
40 ;;;;;;;;;     打印字符串    ;;;;;;;;;;;
41    ;还是用 10h 中断,不过这次调用 13 号子功能打印字符串
42    mov ax, message
43    mov bp, ax                 ; es:bp 为串首地址,es 此时同 cs 一致,
44                               ; 开头时已经为 sreg 初始化
45
46    ; 光标位置要用到 dx 寄存器中内容,cx 中的光标位置可忽略
47    mov cx, 5                  ; cx 为串长度,不包括结束符 0 的字符个数
48    mov ax, 0x1301             ;子功能号 13 显示字符及属性,要存入 ah 寄存器,
49                               ; al 设置写字符方式 ah=01: 显示字符串,光标跟随移动
50    mov bx, 0x2                ; bh 存储要显示的页号,此处是第 0 页,
51                               ; bl 中是字符属性,属性黑底绿字(bl = 02h)
52    int 0x10                   ; 执行 BIOS 0x10 号中断
53 ;;;;;;;;;     打字字符串结束    ;;;;;;;;;;;;;;;
54
55    jmp $                      ; 使程序悬停在此
56
57    message db "1 MBR"
58    times 510-($-$$) db 0
59    db 0x55,0xaa
```

简短说一下代码功能，在屏幕上打印字符串“1 MBR”，背景色为黑色，前景色为绿色。

由于还没有给大家讲解显卡的使用方法，故本段代码中关于“打印显示”的操作都利用 BIOS 给我们建立好的例程就好了，这里第 0x10 号中断便是负责有关打印的例程。

0x10 中断是最为强大的 BIOS 中断了，调用的方法是把功能号送入 ah 寄存器，其他参数按照 BIOS 中断手册的要求放在适当的寄存器中，随后执行 int 0x10 即可。我们不用太细致琢磨 BIOS 功能调用了，大家可以参数代码中的注释了解下即可，毕竟咱们这里用 BIOS 中断只是临时的，以后也用不到了。

第 3 行的“vstart=0x7c00”表示本程序在编译时，告诉编译器，把我的起始地址编译为 0x7c00。

第 4～8 行是用 cs 寄存器的值去初始化其他寄存器。由于 BIOS 是通过 jmp 0：0x7c00 跳转到 MBR 的，故 cs 此时为 0。对于 ds、es、fs、gs 这类 sreg，CPU 中不能直接给它们赋值，没有从立即数到段寄存器的电路实现，只有通过其他寄存器来中转，这里我们用的是通用寄存器 ax 来中转。例如 mov ds：0x7c00，这样就错了。

第 9 行是初始化栈指针，在 CPU 上运行的程序得遵从 CPU 的规则，mbr 也是程序，是程序就要用到栈。目前 0x7c00 以下暂时是安全的区域，就把它当作栈来用。

第 11～28 行是清屏。因为在 BIOS 工作中，会有一些输出，如检测硬件的结果信息。为了让大家看清楚我们在 MBR 中的输出字符串，故先把 BIOS 的输出清掉，这里演示的是 BIOS 中断 int 0x10 的用法。

第 30～35 行是做打印前的工作，先获取光标位置，目的是避免打印字符混乱，覆盖别人的输出。其实这是防君子不防小人的做法，万一别人不在光标处打印，自己打印的内容同样也会被别人覆盖。不管别人了，咱们做好自己的就行，老老实实地只在光标处打印。不知道这是否能提醒大家，字符打印的位置，不一定要在光标处，字符的位置只和显存中的地址有关，和光标是没关系的，这只是人为地加个约束，毕竟光标在视觉上告诉了我们当前字符写到哪里了，完全是为了好看，不要以为光标就是新打印字符的位置。更多细节，以后讲显卡时会提到。

这里还用到了页的概念，您看第 33 行，往 bh 寄存器中写入了 0，这是告诉 BIOS 例程，我要获取第 0 页当前的光标。什么是页呢？

显示器有很多种模式，如图形模式、文本模式等，在文本模式中，又可以工作于 80*25 和 40*25 等显示方式，默认情况下，所有个人计算机上的显卡在加电后都将自己置为 80*25 这种显示方式。80*25 是指一屏可以显示 25 行、每行 80 列的字符，也就是 2000 个字符。但由于一个字符要用两字节来表示，低字符是字符的 ASCII 编码，高字节是字符属性，故显示一屏字符需要用 4000 字节（实际上，分配给一屏的容量是 4KB），这一屏就称为一页，0 页是默认页。

第 38～52 行是往光标处打印字符。说一下第 48 行的 mov ax，0x1301，13 对应的是 ah 寄存器，这是调用 0x13 号子功能。01 对应的是 al 寄存器，表示的是写字符方式，其低 2 位才有意义，各位功能描述如下。

（1）al=0，显示字符串，并且光标返回起始位置。

（2）al=1，显示字符串，并且光标跟随到新位置。

（3）al=2，显示字符串及其属性，并且光标返回起始位置。

（4）al=3，显示字符串及其属性，光标跟随到新位置。

第 55 行执行了个死循环，$是本行指令的地址，这属于伪指令，是汇编器在编译期间分配的地址。在最终编译出来的程序中，$会被替换为指令实际所在行的地址。jmp 是个近跳转，$是 jmp 自己的地址，于是跳到自己所在的地址再执行自己，又是跳到自己所在的地址再继续执行跳转，这样便实现了死循环。可见 CPU 可乖了，它只会埋头做事，并不会觉得有什么不妥，靠谱，值得依赖。

第 57 行是定义打印的字符串。

第 58 行的$$是指本 section 的起始地址，上面说过了$是本行所在的地址，故$-$$是本行到本 section 的偏移量。由于 MBR 的最后两个字节是固定的内容，分别是 0x55 和 0xaa，要预留出这 2 个字节，故本扇区内前 512-2=510 字节要填满，那到底要用多少字节才能填满此扇区呢。用 510 字节减去上面通过$-$$得到的偏移量，其结果便是本扇区内的剩余量，也就是要填充的字节数。由此可见第 50 行的“times 510-（$-$$） db 0”是在用 0 将本扇区剩余空间填充。

代码说完了，可还有两件大事要做，1 是编译，2 是如何将编译后的文件存储到 0 盘 0 道 1 扇区中成为 MBR，以供 BIOS 大神加载之用。

前面介绍了 nasm 的用法，咱们马上来编译汇编代码。

nasm -o mbr.bin mbr.S 回车，您看，这样就编译成功了，我连-f 都没有指定吧。按理说此文件大小是 512 字节，咱们用 ls 命令验证一下：ls -lb mbr.bin 回车，以下是 ls 的输出。

```
-rw-rw-r--. 1 work work 512 7月  26 21:10 mbr.bin
```

用过 Linux 的同学对这个输出还是很熟悉的，若头一次用 Linux 的同学也不要慌张，这里面好多的信

息并不重要，只要看看中间部分就好了，512，果然是 512 字节，这下心里踏实了，下一步是考虑如何将此文件写入 0 盘 0 道 1 扇区。

这里再给大家介绍另一个 Linux 命令：dd。dd 是用于磁盘操作的命令，功能太强大了，有如穿甲弹一样，可以深入磁盘的任何一个扇区，无坚不摧。所以，它也可以删除 Linux 操作系统自己的文件，是把双刃剑。

还是先看帮助文件，man dd 回车，为了节约大家的时间，我只把咱们今后用到的几个选项摘了出来，还是那句话，够用就行了，需要时再学。

```
if=FILE
read from FILE instead of stdin
```

此项是指定要读取的文件。

```
of=FILE
write to FILE instead of stdout
```

此项是指定把数据输出到哪个文件。

```
bs=BYTES
read and write BYTES bytes at a time (also see ibs=,obs=)
```

此项指定块的大小，dd 是以块为单位来进行 IO 操作的，得告诉人家块是多大字节。此项是统计配置了输入块大小 ibs 和输出块大小 obs。这两个可以单独配置。

```
count=BLOCKS
copy only BLOCKS input blocks
```

此项是指定拷贝的块数。

```
seek=BLOCKS
skip BLOCKS obs-sized blocks at start of output
```

此项是指定当我们把块输出到文件时想要跳过多少个块。

```
conv=CONVS
convert the file as per the comma separated symbol list
```

此项是指定如何转换文件。

```
append append mode (makes sense only for output; conv=notrunc suggested)
```

这句话建议在追加数据时，conv 最好用 notrunc 方式，也就是不打断文件。

齐了，dd 的介绍就到这了，赶紧试验一下这个神奇的工具吧。

```
dd if=/your_path/mbr.bin of=/your_path/bochs/hd60M.img bs=512 count=1  conv=notrunc
```

各位看官，请将上面命令行中的 your_path 替换为您自己的实际路径。

输入文件是刚刚编译出来的 mbr.bin，输出是我们虚拟出来的硬盘 hd60M.img，块大小指定为 512 字节，只操作 1 块，即总共 1*512=512 字节。由于想写入第 0 块，所以没用 seek 指定跳过的块数。

执行上面的命令后，会有如下输出。

记录了 1+0 的读入

记录了 1+0 的写出

512 字节（512 B）已复制，0.313312 秒，1.6 KB/秒

这就说明命令执行成功了，mbr.bin 已经写进 hd60M.img 的第 0 块了。借鉴美国宇航员阿姆斯特朗的一句话：虽然这只是简单的一小步，但却是实现我们自己系统的一大步。记得当初我可是非常激动呢。

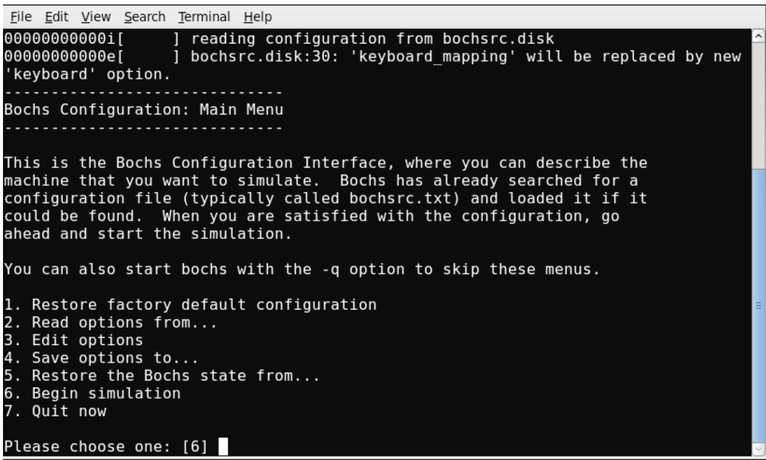
启动 bochs 测试一下，我习惯到 bochs 安装目录下启动它，bin/bochs –f bochsrc.disk 回车，接着会显示如图 2-4 所示的界面。

默认是[6]，开始模拟啦。回车。

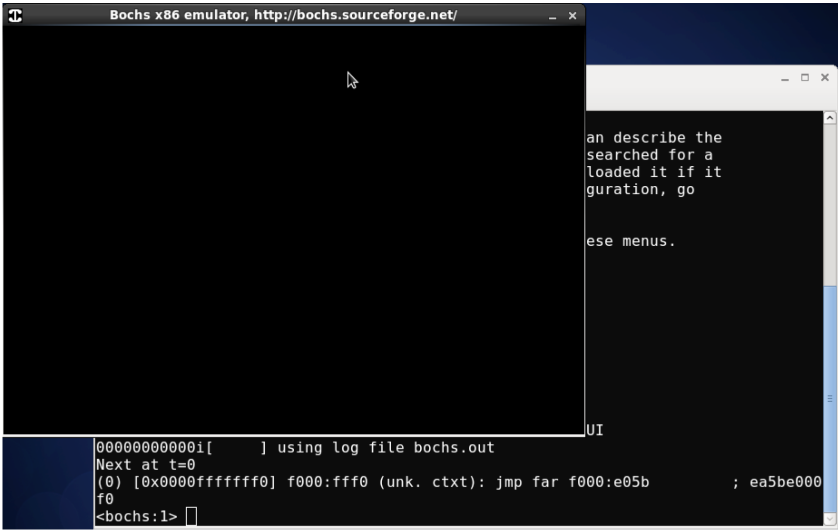
由于咱们编译的是可调试的版本，所以会停下来，bochs 等待咱们键入下一步的命令，如图 2-5 所示。

大家看到，这一下弹出了两个界面，前面的那个是 bochs 所模拟的机器，可以认为它就是台电脑了，不仅仅是电脑的显示器。后面的界面是 bochs 的控制台，咱们控制 bochs 运行就要在这里输入命令。现在激活后面

的 bochs 控制台，输入字符 c 后，回车。bochs 所模拟的机器就开始运行了。这里键入的 c 是 continue，调试方法同 gdb 类似，详细的 bochs 操作方法咱们会在下一章中介绍。

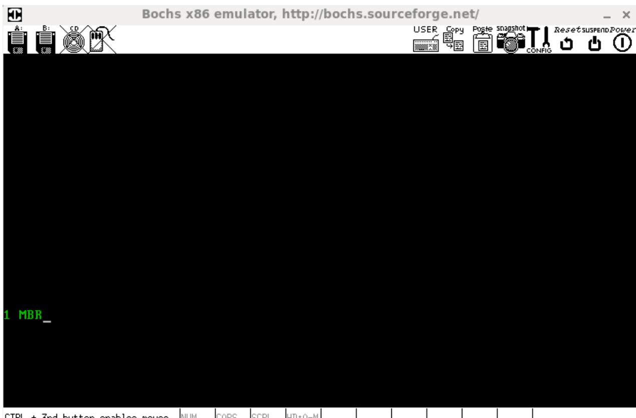


▲图 2-4　bochs 启动



▲图 2-5　bochs 运行

MBR 运行起来后，就会出现下面的效果，如图 2-6 所示。



▲图 2-6　bochs 运行 mbr

下一章正式开讲细节部分。

第 3 章　完善 MBR

上一章我们完成了 MBR 的雏形，虽然有些简陋，但那可是我们的一大步，我们已经完全占领了计算机，可以在自己的空间里自由驰骋，想干嘛都行了。也许某些同学还沉浸在喜悦之中，心想，这下终于可以一展身手了。不过有的同学还是有些小闹心，心说，虽然给了我一片“疆土”，但我能拿它们干吗，我连骑马还不会呢，叫我如何在这片土地上印上我的足迹。不过没关系，我在上一章中答应大家要细说一下 MBR 中的几个细节，说到做到，闲话少说，开工啦。

3.1 地址、section、vstart 浅尝辄止

还记不记得我们的 mbr 程序开头的那句话：SECTION MBR vstart=0x7c00。这里我想跟大家介绍一下 vstart，也许大家觉得上一章不是介绍过了吗？不就是把地址编译为 0x7c00 吗？是，这句话是我说的，站在我的位置我甚至觉得它描述得无比精确，但我担心未必把它说清楚了，对于头一次听说此概念的同学来说，vstart 不是那么简单的，“把地址编译为 0x7c00”这句话仅仅是水面上的冰山，这水下看不到的部分可大着呢。

咱们多花点时间，大家随我一起再深入学习一下吧。

我当初学习时汇编语言时，对 vstart 很模糊，想问问题吧，又不知道问什么，似乎是不知道哪里不会，不知从何问起。其实是哪里都不懂，根本没掌握，能问出问题的同学才是掌握得差不多了。按理说，vstart 是 section 中的概念，所以要先学习下 section 才行，但要理解 section 得先理解地址是什么才行，咱们处处体现着递归式学习，遇到不明白的就深入进去搞清楚。

所以，我觉得地址的概念都没搞清楚的话，section 和 vstart 这两个高级的玩意学起来当然如空中楼阁了。在这里我把当初那会的学习体会在这里跟大家分享一下，这里用了几个例子来演示，希望别把大家搞得更迷糊，嘿嘿，我努力说清楚。

3.1.1　什么是地址

地址只是数字，描述各种符号在源程序中的位置，它是源代码文件中各符号偏移文件开头的距离。由于指令和变量所占内存大小不同，故它们相对于文件开头的偏移量参差不齐。源文件就像旅店一样，里面的符号（指令、变量等）就像旅店里的房间，有单人间、双人间，虽然大小不同，但它们也需要被旅店管理员编号，也就是每个房间都有房间号，这样房客通过房间号便能找到自己的住所。房间由旅店管理员给编址，那源代码文件中各符号地址又是由谁来规划的呢？

编译器的工作就是给各符号编址。编译器根据所在硬件平台的特性，将源代码中的每一个符号（指令和数据）都按照本硬件平台的特性分配空间，在不考虑对齐的情况下，这些符号在空间上都彼此相邻，连续分布，它们在程序中距第一个符号的距离便是它们在程序中的地址。

这么说还是有点抽象，地址确实是很抽象的东西，看不到，摸不着，不过学习起来完全靠想像力可不成，那就别那么费劲了，咱们拿实际程序说事。先跟大家交待一下，下面所说的程序是纯二进制可执行文件，这样方便解释，其他类型文件牵扯到文件头，其实道理是一样的，只不过描述起来不方便。

本质上，程序中各种数据结构的访问，就是通过“该数据结构的起始地址＋该数据结构所占内存的大小”来实现的。题外话，这就解释了为什么定义变量要给出变量类型，因为变量类型规定了变量所占内存大小，每种类型都有其对应的内存容量。

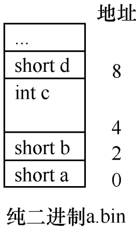
该数据结构的起始地址是怎样得到的呢？

程序中定义的任何一个变量，在编译之后的可执行文件中都会占据一席之地。此变量在文件中的位置是编译器来安排的。编译器是人设计的，所以如果是您来设计编译器，您会怎样规划其地址呢？

按理说，人只能创造出人的思维能想到的东西，所以无论人创造出什么，一定能够被人来理解，无非是时间长短的问题。

编译器无论怎样安排程序中的数据，必然有个先后顺序，而占据第一位的数据，其地址便是整个程序的起始地址，在它后面的其他数据依次排开就行了。若以整个程序的开头部分为基准，它的第一个数据所在的位置必然也是在文件的开头，也就是偏移文件开头为 0 的地方。第二个数据所在的位置是数据 1 的起始（偏移为 0）+数据 1 所占的内存大小。第 n 个数据所在的位置便是数据 n-1 的偏移+数据 n-1 的内存空间。可见，数据的地址，其实就是该数据相对整个程序开头的距离，即偏移量。又一个题外话，由于程序第 1 个数据的地址（偏移量）是 0，所以数组中第 1 个元素的下标也是 0，本书也是从第 0 章开始的，可见偏移量的概念太深入人心了。

为了说明地址是偏移量的概念，咱们由浅入深，先拿变量举个例子，如图 3-1 所示。



▲图 3-1　地址

变量 a 是短整型，其所占内存大小为 2 字节，其偏移量为 0。变量 b 是短整型，也是占 2 字节空间，其偏移量为 2。变量 c 是整型，占 4 字节空间，其偏移量为 4。变量 d 是短整型，占 2 字节空间，其偏移量为 8。

上述的偏移量本质上就是地址，每个变量的地址是前一个变量的地址+前一个变量的内存空间大小。

看完了这个简单的例子，咱们要上点开胃菜了，还是拿实际代码解释更有说服力。

代码编译之后，源代码中的标号会被替换为实际地址，因为这些就是编译器提供的标号以及伪指令，只是方便程序员写代码用的，并不是 CPU 支持的东西，所以编译之后，这些伪指令都被转换为 CPU 能够识别的东东了。

若想看到程序编译后的各指令或标号的地址，可以用反汇编来查看。虽然方法算不上高大上，如果大家有更好的办法的话，还请您告诉我一声，兄弟提前谢谢各位啦。

代码 3-1 前两列是源码和相应的行号，后三列是反汇编出来的地址、内容（机器码或数据）和汇编代码。我是用 ndisasm 这个工具完成的，大家有兴趣可以自行查阅相关资料，简单易用。

代码 3-1

行号	源码	地址	地址处的内容：机器码或数据	反汇编代码
1	mov ax,$$	00000000	B80000	mov ax,0x0
2	mov ds,ax	00000003	8ED8	mov ds,ax
3	mov ax,[var]	00000005	A10D00	mov ax,[0xd]
4	label: mov ax,$	00000008	B80800	mov ax,0x8
5	jmp label	0000000B	EBFB	jmp short 0x8
6	var dw 0x99	0000000D	99	
7		0000000E	00	

首先这段程序并没有什么实际意义，大家不要琢磨代码功能了，这里只是为了帮助大家理清地址是什么。

源码经编译之后都会变成我们不认识的东西，这个东西是机器码，还是数据，取决于源码中相应的内容。例如第二行“mov ds,ax”，明显是指令，被编译器处理之后，其所在地址处的内容就是机器码 8ED8，长得完全不像“mov ds,ax”了。

第六行的源码“var dw 0x99”，明显就是数据定义，查看内容列，它经编译之后，变成了 99、00，因为 dw 是定义一个字，双字节，并且 x86 是小端字节序，低位的 99 在低地址，高位的 00 在高地址，所以地址 D 处内容为 99，地址 E 处内容为 00。

为了感谢编译器为咱们做的贡献，简要说下复杂的机器指令。

不知大家有没有感觉到奇怪，第 1 行和第 4 行的 mov 操作，机器码第 1 个字节都是 B8，而另外第 2、3 行同样是 mov 指令，机器码却有天壤之别，似乎找不到共性。原因是机器码是由很多部分组成的，这个说起来就有点复杂了，比如指令前缀、主操作码字节以及寻址方式字节。寻址方式由 ModR/M 字节、SIB

字节、位移量、立即数组成。第 1 行和第 4 行的 mov 的机器码中第 1 字节都是 B8，其原因是寻址方式都是立即数。第 5 行的“无条件跳转 jmp”对应的机器码第 1 字节是 EB。还有“小于时跳转 jl”,“等于时跳转 je”,“不小于等于时跳转 jnle”……“跳转家族”成员多着呢，有没有感觉到机器码好乱，好麻烦，是不是累觉不爱了……不过大家不用琢磨这个机器码了，要是把它搞清楚了还要汇编语言有啥用，咱直接写机器指令算啦。大家若对此感兴趣，可以参考指令格式。

看第 1 行的 mov 指令，$$表示的是所在的 section 的起始地址，由于这 6 行代码中没有定义 section，故 nasm 默认把全体文件当成一个大的 section，全体文件自然偏移地址为 0，所以在反汇编代码那列中，起始地址$$被置换为 0。

第 2 行代码是真指令，不牵涉到符号转换，所以反汇编后的代码同源码一致。

第 3 行引用了 var 变量的值，[]符号是取所在地址处的内容。在相应的反汇编代码中，相应的第三行中 var 这个符号地址被编译器替换为 0xd。结合地址列查看一下内容列，地址为 0xd 的内容为 99，这正是 var 的值。

第 4 行源码为“label: mov ax,$”,label 是个标号，代表指令“mov ax，$”所在地址。$是个隐式的标号，表示当前行地址。按理说这两个标号值应该是一致的，验证一下：查看下反汇编代码列的第 4 行，$被替换为 0x8，即本行 mov 指令地址是 0x8，在地址列第 4 行查看，地址确实为 8，吻合。

第 5 行的“jmp label”编译后被替换为 jmp short 0x8，这是短跳转指令，地址为 8 处的内容是第 4 行的“mov ax，$”，同样吻合。

第 6 行的便是数据定义了，定义了双字节变量 var，其值为 99。在内容处的第 6 行可知，内容为 99，与源码定义吻合。

顺便说一句，CPU 不去判断给它的内容是指令，还是一般数据，它也分不清楚。CPU 执行指令时是顺次向下执行的，所以若没有第 5 行的 jmp 形成死循环，CPU 执行到第 6 行时，会把 var 变量的值 99 当成指令，是否会报错还不得而知，这得看给它的数是否恰好能成为个指令。由上面提到的指令格式可知，指令有很多组成部分，有的组成部分还可以省略，万一给的这个数字恰好能组成一个指令呢。如此处 var 变量值 99 就恰好是汇编语言中的字扩展指令 CWD，它的功能是将一个字型变量扩展为双字型变量，即 Change Word to Double word。

终于说到重点了，大家观察一下，“地址”列中的数字和“内容”列中的内容有这样一种关系：地址等于上一个地址+上一个地址处的内容的长度。例如地址列第二行的 3 等于“上一个地址 0”+“上一个地址 0 处的内容：B80000 的长度 3”，以此类推。

这说明这和咱们前面的示意图是一致的。

于是，现在可以斩钉截铁地得出结论：编译器给程序中各符号（变量名或函数名等）分配的地址，就是各符号相对于文件开头的偏移量。

3.1.2 什么是 section

不知道有多少同学对汇编语言中的 section 感到迷惑，这个 section 称为节，在有的编译器中，同时支持 segment 和 section 这两个关键字，它们的功能都是在程序中宣称一个区域。不过这可不是 Linux 用户进程中的段。为不引起混乱，咱们只用 section 来描述汇编语言中的段。

编译器提供的关键字 Section 只是为了让程序员在逻辑上将程序划分成几个部分，因为它是伪指令，CPU 都不知道有这个东西，更不知道咱们交给它执行的代码经过了这很多的“风风雨雨”。甚至，我怀疑 nasm 即使提供了这个 section，它也不知道这个 section 中的内容是什么，是代码？数据？栈？nasm 不关心，也没必要关心，因为这是它给程序员的福利，程序员自己知道在哪个 section 中是什么就行啦。一般 section 的应用场所是根据不同的属性人为地将程序划分几部分，如数据放在一个 section 中，指令放在另一个 section 中，这样程序员便将指令和数据分开了，使代码结构清晰明了，更易于维护。程序如何划分，这个没有规定，完全是看程序员自己的风格喜好，甚至可以利用 section 把程序切得零碎不堪，所以你懂了，nasm 根本没必要知道你的 section 中到底是啥。

现在咱们看看，在程序加入 section 后，编译器为我们带来了哪些改变。

代码 3-2

行号	源码	地址	地址处的内容：机器码或数据	反汇编代码
1	section code			
2	mov ax,$$	00000000	B80000	mov ax,0x0
3	mov ax,section.data.start	00000003	B81000	mov ax,0x10
4	mov ax,[var1]	00000006	A11000	mov ax,[0x10]
5	mov ax,[var2]	00000009	A11400	mov ax,[0x14]
6	label: jmp label	0000000C	EBFE	jmp short 0xc
7		0000000E	0000	
8	section data			
9	var1 dd 0x4	00000010	0400	
10		00000012	0000	
11	var2 dw 0x99	00000014	99	
12		00000015	00	

代码 3-2 在上一个基础上稍做了一些改变，添加了变量及 section 的应用。

第 1 行和第 8 行为空，并没有产生相应地址或机器指令，这就是 section 是伪指令的原因。

第 3 行中用到了 section.data.start，其用法是 section.节名.start，这里是获得名为 data 的 section 在本文件中的真实偏移，即起始地址，是 nasm 提供的伪指令。查看“反汇编代码”列的同一行，编译后已经被替换为 0x10，说明定义的数据 section 起始地址为 0x10。可见，定义的 section，其起始地址默认是从上一个数据的地址延续下来的。

由于 var1 是数据 section 的首个数据，其地址必然是和数据 section 一致。故对比下“源码”列和“反汇编代码”列的第 4 行，var1 的地址确实被替换为 0x10。

var2 是继 var1 之后的变量，由于 var1 的类型是 dd，即双字，故其占用 4 字节，所以 var2 的地址应该是 0x14。这一点可以通过对比“源码”列和“反汇编代码”列看出。

于是，现在可以趾高气扬地得出结论：关键字 section 并没有对程序中的地址产生任何影响，即在默认情况下，有没有 section 都一个样，section 中数据的地址依然是相对于整个文件的顺延，仅仅是在逻辑上让开发人员梳理程序之用。

3.1.3　什么是 vstart

先看看官方的说法，以下摘自 nasm 技术手册，上面给出了它的名称和作用。

nasm 手册：

7.1.3 Multisection Support for the bin Format

- Sections can be given a virtual start address，　which will be used for the calculation of all memory references within that section with vstart=.

大概意思是 section 用 vstart=来修饰后，可以被赋予一个虚拟起始地址 virtual start address（强调了这个是虚拟的地址，不过要注意，这与 x86 CPU 中开启分页后的虚拟地址是两码事，不能混为一谈），它被用来计算在该 section 内的所有内存引用地址。

vstart 是虚拟起始地址，这里面有两个重要的概念，1 虚拟，2 起始。

以下我要通过“长篇大论”来解释一遍，各位做好心理准备。

vstart 的作用是为 section 内的数据指定一个虚拟的起始地址，也就是根据此地址，在文件中是找不到相关数据的，是虚拟的，假的，文件中的所有符号都不在这个地址上。

给大家再整一个硬菜，继续看代码。

代码 3-3

行号	源码	地址	地址处的内容：机器码或数据	反汇编代码
1	section code vstart=0x7c00			
2	mov ax,$$	00000000	B8007C	mov ax,0x7c00
3	mov ax,section.code.start	00000003	B80000	mov ax,0x0
4	mov ax,section.data.start	00000006	B81400	mov ax,0x14
5	mov ax,$	00000009	B8097C	mov ax,0x7c09
6	mov ax,[var1]	0000000C	A10009	mov ax,[0x900]
7	mov ax,[var2]	0000000F	A10409	mov ax,[0x904]
8	jmp $	00000012	EBFE	jmp -2
9	section data vstart=0x900			
10	var1 dd 0x4	00000014	0400	
11		00000016	0000	
12	var2 dw 0x99	00000018	99	
		00000019	00	

样例代码 3-3 在 section 中添加了 vstart，这个参数是让编译器将 section 中的数据的地址以 vstart 的值为起始，不再从整个程序开头算起。只有以程序开头 0 算起的地址才是真实存在的，在这个地址上能访问到相应的符号，所以不以程序开头算起的地址，必然在程序内部不存在，是虚拟的。

源码第 1 行的 section 含有 vstart=0x7c00，故该节中的数据地址以 0x7c00 为起始编址。此 0x7c00 便是虚拟的地址，在程序中没有偏移文件开头为 0x7c00 的地址，整个程序不到 100 字节。

源码第 2 行的$$在编译后被替换为 vstart 的值 0x7c00，可见，$$以该节的虚拟起始地址为主，若该节未用 vstart 来指定则以在文件中的起始地址（较虚拟地址来说，此地址便为真实地址）为主，而该节在文件中的起始地址是 code.节名.start。

第 3 行和第 4 行的源码和相应反汇编代码表明“code.节名.start”是节在整个程序中的地址，即相对于文件开头的偏移量。

第 5 行的$在文件中的地址是 0x9，经编译后变成了 0x7c09，类似于重定位：新的地址+在文件中的地址（也相对于整个文件的偏移量），即 0x7c00+9。

第 6、7 行中引用的变量 var1 和 var2 属于 data 节，由于该节有 vstart=0x900，所以该节中的 var1 地址是 0x900，var2 是 0x900+var1 的内存空间 4 字节=0x904。这里的 0x900 也属于虚拟地址，原因同 0x7c00 一样。

这里要和大家说一下，第 8 行的 jmp $，按理说可以编译成相对转移的形式：jmp -2，而我用 ndisasm 反汇编的结果是 jmp short 0x12，0x12 是相对于整个文件的绝对地址，可称为真实地址。此处的反汇编结果是我手工修改成“jmp -2”的，原因如下。

我心想，既然我已经用 vstart 指定虚拟起始地址了，jmp $ 编译的结果按理说要么是 jmp near 0x7c12，操作码以 E9 开头，这说明这是相对近转移的语法：jmp 16 位地址，要么是 jmp short -2，操作码是 EB，这是相对短转移的语法：jmp -128～127 之间的数。可是看这 jmp short 0x12，0x12 是真实的地址，似乎不正确。鉴于 jmp short 是相对短转移，其操作码是 EB，占 1 字节，操作数是相对于跳转目标地址的偏移量，是 1 字节的有符号数，故可正可负，只能在段内跳转。于是我用 ndisasm 反汇编时用参数-o 0x7c00 给其添加了个起始地址来测试，如命令“ndisasm –o 0x7c00”，但反汇编的结果变成了 jmp short 0x7c00，这结果明显不对，偏移量 0x7c00 不在-128～127 之间。由于 jmp short 指令操作码和操作数各是 1 字节，总共加起来是 2 字节。于是怀着忐忑的心情在 bochs 中验证了下，执行到 jmp $的时候果然是 jmp -2。所以上面第 8 行是被我手工改为 jmp -2 的，请大家知晓。

地址访问策略是根据程序中给出的地址，到地址处去拿东西，所以这个东西要提前在那个地址处准备好了才行，这是永恒不变的。程序中给出的地址，最终是要用来访问物理内存的，所以该地址对应的物理内存必须是你想要的才行，换句话说，必须保证自己想要的东西被加载到那个物理内存位置。这就是程序加载器做的事，根据文件头中给出的各段的位置，将它们加载到内存中的相应地址，这样用户程序才能访

问到自己需要的东西。

vstart=xxxx 和 org xxxx 这两个关键字是同一功能，但很多同学都混淆其意义。它们并不是告诉编译器程序加载到地址 xxxx，这似乎一听，好像是编译器有加载器的功能了，“加载”不是它的工作，这是加载器的工作，编译器只会规划代码，只会为程序编址，并不负责加载。

vstart 和 org，它们的功能是告诉编译器：“嘿，老兄，你帮我把后面所有数据（指令和变量）的地址以 xxxx 为起始开始编吧”。就这点意思而已，没别的了。编译器从不问你这是为啥。只有开发人员知道以 xxxx 为起始的原因是将来有某个加载器要把我的这个程序放到内存的 xxxx 地址，如果我程序中引用的所有地址不是以 xxxx 为起始的，那就坏了，访问错的数据肯定出事。

再重复一次，编译器只负责编址，它只会将数据相对于文件开头的偏移量作为该数据的地址，全是以 0 起始的。所以把整个程序加载到内存地址 0 处，程序运行肯定是没问题的。但基本上没有哪个程序能有如此待遇在 0 物理地址上走上一回（不包括保护模式下分页产生的虚拟地址）。

编译器以相对于文件开头偏移来编址的好处是利于重定位。整个文件新加载到某个地址后，可以以这个地址为段基址，文件内的数据的地址是以 0 为开始算的，所以它们直接就可以用作段内偏移地址了。

为什么 mbr 能够运行正常？

mbr 用 vstart=0x7c00 来修饰的原因，是因为开发人员知道 mbr 要被加载器（BIOS）加载到物理地址 0x7c00，mbr 中后续的物理地址都是 0x7c00+。另外，因为 BIOS 进入 mbr 是通过 jmp 0：7c00 来实现的，故此时 cs 已经变成 0，相当于“平坦模型”了，只不过此“平坦模型”大小只是 65536 字节，而不是 4GB。所以 mbr 中各数据编译出来的地址（大于等于 0x7c00）实际上都成了偏移地址，这样“0*16：偏移地址 0x7c00+”来访问被加载到 0x7c00 的 mbr 是正确无误的。所以说，用 vstart 的时机是：我预先知道我的程序将来被加载到某地址处。程序只有加载到非 0 地址时 vstart 才是有用的，程序默认起始地址是 0。

其实 section 用 vstart=0 修饰更方便，节内元素地址是以 0 为起始的偏移，由于实模式下会使段寄存器中的值乘以 16，所以咱们提供的段基址要事先除以 16，如使 ds=（section.xxx.start+加载的地址）/16，这样访问节内数据直观多了。注意，在保护模式下段寄存器中的值是不用乘以 16 的，在后面介绍实模式和保护模式时会给大伙儿解释明白。

举个生活中的例子：比如我想去找李雷，在李雷不外出的情况下，直接去李雷家就行了。但李雷上午跟我说，下午韩梅梅叫他去学校了，找他的话直接去学校吧。

这个简单的例子就是模拟的 CPU、访问对象、加载器三者的关系。

这里面我相当于 CPU，李雷相当于程序中访问的数据，韩梅梅相当于加载器。李雷知道下午会被韩梅梅叫到学校，因为他的地址变了，所以让我改变了找他的地点。由于程序知道自己将来会被加载器放到某个地址，所以告诉编译器把它编成这个地址，将来 CPU 用此地址才能找到它。

再提醒一下，vstart 只是告诉编译器以新的数字作为后面数据的地址的起始值，它本身没改变数据本身在文件中的地址（相对于文件开头的偏移），若能改的话，比如 vstat=0x7c00，那 0x7c00 之前的间隙肯定得用 0 来填充，那得多出多少填充物啊，那文件就太大啦。总不该用了 vstart 后，文件就跟着变大。

由上分析，vstart 只是按照开发人员的意愿安排新的起始地址，不再以文件开头 0 为起始，其地址若超过文件大小则不会落在文件内，所以是虚拟的。

不管程序用的是不是虚拟地址，只要交给地址总线一个地址，地址总线就会去寻找该地址处的内容。根据这个原则，只要保证该地址处的内容正是你所需要的即可。如果程序员用 vstart 指定了新的地址，干涉了编译器编址的方式，程序员要清楚地知道自己需要的东西是否会出现在物理内存中这个新的地址处。

3.2 CPU 的实模式

由于 mbr 在实模式下工作……什么？什么是实模式？这时候有同学打断了我。我心想，这下好办了……哈哈，没有啦，开个玩笑而已。我们这里所说的实模式其实就是 8086 CPU 的工作环境、工作方式、工作状态，这是一整套的内容，并不是单指某一方面的设置。

实模式是指 8086 CPU 的寻址方式、寄存器大小、指令用法等，是用来反应 CPU 在该环境下如何工作的概念。所以想了解实模式这种抽象的概念，主要就是了解在实模式下 CPU 能做什么。

大家都学过汇编语言吧，里面有讲实模式、保护模式之类的，但鉴于太过久远，为了让后面的工作顺利进行，我觉得还是有必要给大家介绍下 CPU 的工作模式。

3.2.1　CPU 的工作原理

在介绍 CPU 的各种模式之前，先占用大家几分钟的时间，说点在两种模式下公共的内容，和大家聊聊 CPU 的工作原理。

当然这里所说的工作原理可不像微机接口技术里那么细致，精确到逻辑门等电子电路。第一我也不会，想讲也讲不出来。第二我觉得没必要懂到那么细致，如果懂的太细了，会为之所累，您想，每次执行一条指令时，您的大脑总是联想到各种元件的工作流程，本来一瞬间完成的工作您可能要给放大一千倍，非得强迫症似的要求掌握每个步骤的细节，万一在哪个点上想不通了这就会让人感到很沮丧，影响心情，甚至质疑上层的编译器，哈哈，我知道说得有点严重了，不夸张一点的话不容易表述问题，好了，下面在宏观上介绍下 CPU 工作原理。

大家都知道，CPU 的唯一的任务就是执行指令，在它眼里，指令就是一串 010101……那它执行每条指令的流程是怎样的呢？

CPU 大体上可以划分为 3 个部分，它们是控制单元、运算单元、存储单元。

控制单元是 CPU 的控制中心，CPU 需要经过它的帮忙才知道自己下一步要做什么。而控制单元大致由指令寄存器 IR（Instruction Register）、指令译码器 ID（Instruction Decoder）、操作控制器 OC（Operation Controller）组成。程序被加载到内存后，也就是指令这时都在内存中了，指令指针寄存器 IP 指向内存中下一条待执行指令的地址，控制单元根据 IP 寄存器的指向，将位于内存中的指令逐个装载到指令寄存器中，但它还是不知道这些指令是什么，在它眼里的 0101 串此时还没有实际意义。然后指令译码器将位于指令寄存器中的指令按照指令格式来解码，分析出操作码是什么，操作数在哪里之类的。表 3-1 给出了一般的指令格式。

表 3-1　　IA32 指令格式

前缀	操作码	寻址方式、操作数类型	立即数	偏移量

看上去还是很复杂的，复杂的原因是必须用一种统一的格式去归纳所有形式的指令，因为 CPU 只能识别一种格式。

由于 CPU 支持的指令数量较多，一些指令还可以搭配一些辅助的东东，所以就需要在前缀部分记录这些，如 rep（用于重复执行，汇编中经常用）、段超越前缀。操作码就是大家平时用的 mov、jmp 等。寻址方式又有好多，如基址寻址、变址寻址等，操作数类型中记录的是用哪些寄存器之类的。如果在指令中用到了立即数，就要将其记录到指令格式中立即数的部分，如果寻址方式中用到了偏移量，就要将此偏移量记录到指令格式中的偏移量部分。

既然指令是存放在指令寄存器中的，那指令中用到的数据存放到哪里呢？下面介绍存储单元。

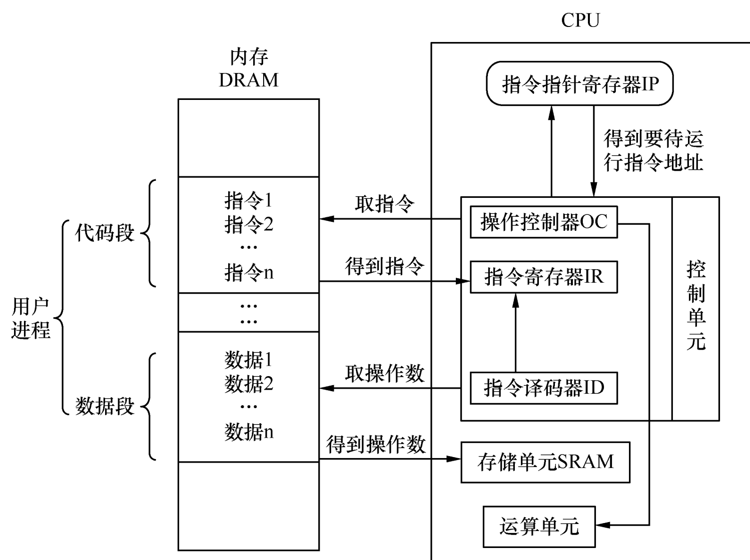
存储单元是指 CPU 内部的 L1、L2 缓存及寄存器，待处理的数据就存放在这些存储单元中，这里的数据是说指令中的操作数。为什么数据已经在内存中了还非得在 CPU 内部再整这么个存储单元干吗？原因是缓存基本上都是采用的 SRAM（Static RAM）存储器，从名字上看就知道它是一种具有静态存取功能的存储器。这么一说，似乎还有动态存储功能的存储器？是啊，其实我们插在主板上的物理内存就是 DRAM（Dynamic Random Access Memory），DRAM 内存需要每隔一段时间就去刷新电路，刷新就是指给 DRAM 充电，否则存储的数据就会丢失。而 SRAM 不需要刷新电路即能保存它内部存储的数据，这就是静态的含义，因此 SRAM 性能较强劲。但 SRAM 也不是完美无缺的，它的集成度较低，相同容量之下，SRAM 的体积比 DRAM 要大很多。二级缓存都不大，目前来说顶多 4MB 左右，所以现代 CPU 用二级缓存的数量取胜，如 L1、L2、L3 共三级。寄存器可分为两大类，程序员可以使用的寄存器称为程序可见寄存器，如通用寄存器、段寄存器。程序不可见寄存器是指程序员不可使用，也无法访问到它们，系统运行期间可

能要用到的寄存器，如 ALU 算术逻辑单元在求和时，会将结果先送到数据暂存寄存器。

操作码有了，操作数有了，就差执行指令了，随后“操作控制器”给相关部件发信号，会给哪些部件发信号呢？例如下面要介绍的运算单元。

运算单元负责算术运算（加减乘除）和逻辑运算（比较、移位），它从控制单元那里接收命令（信号）并执行，它没有自主意识，只是个执行部件。

它们之间的关系如图 3-2 所示。



▲图 3-2　CPU 工作原理示意图

好啦，文字描述过了，图也看过了，总结下 CPU 的工作原理。控制单元要取下一条待运行的指令，该指令的地址在程序计数器 PC 中，在 x86CPU 上，程序计数器就是 cs：ip。于是读取 ip 寄存器后，将此地址送上地址总线，CPU 根据此地址便得到了指令，并将其存入到指令寄存器 IR 中。这时候轮到指令译码器上场了，它根据指令格式检查指令寄存器中的指令，先确定操作码是什么，再检查操作数类型，若是在内存中，就将相应操作数从内存中取回放入自己的存储单元，若操作数是在寄存器中就直接用了，免了取操作数这一过程。操作码有了，操作数也齐了，操作控制器给运算单元下令，开工，于是运算单元便真正开始执行指令了。ip 寄存器的值被加上当前指令的大小，于是 ip 又指向了下一条指令的地址。接着控制单元又要取下一条指令了，流程回到了本段开头，CPU 便开始了日复一日的循环，由于 CPU 特别不容易坏，所以唯一它停下来的条件就是停电。

以上是 CPU 的工作原理，无论 CPU 在哪种模式下工作，这一核心原理是不变的。有了这一思想武装起来后再讲模式就简单多了，兄弟们加油，下一节，不见不散。

3.2.2　实模式下的寄存器

先说一下什么是寄存器。

寄存器是一种物理存储元件，只不过它比一般的存储介质要快，能够跟上 CPU 的步伐，所以在 CPU 内部有好多这样的寄存器用来给 CPU 存取数据。

先简短说这一两句，暂时离开一下主题，咱们先看看相对熟悉一些的概念——缓存。

缓存也是一项非常伟大的发明，成功解决了速度不匹配设备之间的数据传输，并且在一般情况下，IO 是整个系统的瓶颈，缓存的出现，有效减少了低速 IO 设备的访问频率，从而大幅度地提升了速度。

大家对缓存一定不会陌生，生活中处处都是缓存的应用。比如浏览器在请求一个域名 ip 时。

（1）浏览器内部都有 dns 客户端，它先查询本地 dns 缓存中是否有该域名的 ip，如果有就直接去访问该 ip。如果没有，该 dns 客户端先要查找自己主机所设置的 dns 服务器，然后去该 dns 服务器去查询 ip。

（2）如果该 dns 服务器本地缓存中有该域名的 A 记录（域名与 ip 地址的对应记录），则直接返回给浏

览器中的 dns 客户端。没有该域名的 A 记录，就通过递归的方式向上询问其他 dns 服务器，也许问到了根 dns 服务器才找到了答案。于是这路上所有被询问过的 dns 服务器，都将此域名对应的 A 记录缓存到自己的 cache 中，以备下次再有相同域名查询时好直接返回。

（3）浏览器中的 dns 客户端得到此域名的 ip 地址后，也将该域名和 ip 放在自己的缓存中，以备下次用户再键入同一域名时，避免再查一次 ip。

（4）浏览器开始通过网络用 http 协议访问该 ip 地址的 80 端口（默认是 80 端口，除非特别指定）。

（5）一般情况下该 ip 对应的设备不是最终的 Web 服务器，很少有人把 Web 服务器直接暴露在公网。假设该 ip 对应的设备是台网关（一般是硬件路由设备），该网关检查本地缓存中是否有相关 Web 服务器的缓存，若有则直接将该 http 请求分配给缓存中的 Web 服务器。否则从服务器列表中重新分配一台 Web 服务器，将该 http 请求转发给该 Web 服务器处理。随后将该 Web 服务器的 ip 地址（一般是内网地址）和端口号缓存起来，以备下次该用户的请求到来时，依然给该 Web 服务器。有的网关可以识别用户 cookie 信息，从而可以将请求再次落到上一个请求的 Web 服务器上。

（6）Web 服务器拿到请求后，如果是静态请求，先检查自己的缓存中是否有该页面的记录，否则直接从硬盘上取出页面，将其返回后，再存入本地静态缓存中。如果动态请求，先交给自己的 cgi 去处理。

（7）cgi 拿到请求后，先检查自己的缓存系统，如 memcache，如果缓存中没有，与数据库建立连接，向数据库发出请求。

（8）数据库也是先检查自己的缓存，若没有结果集，则从表中检索到数据后返回，并将结果集缓存起来。

（9）cgi 拿到数据后，返回给 Web 服务器，并将数据缓存到 memcache 中。

（10）Web 服务器拿到了数据后，将数据返回给网关。由于是动态数据，不需要缓存。

（11）网关拿到数据后，直接返回给浏览器。

（12）如果浏览器发现其中有静态数据，如图片，也将静态数据缓存到用户的 internet 临时目录。

您看，就这么一个不起眼的网页请求，就经过了 12 步，几乎每一步都要缓存结果，当然每一层的缓存都有一定的失效时间，超过多少秒后就将缓存中的记录置为无效。这其中还有另外一个术语，如果缓存中恰好有需要的内容，这就称为命中 hit，否则称为缺失 mis。

其实上面的缓存，我还没说全呢，硬件之间还有缓存呢，比如硬盘控制器和硬盘……得，您还是打住吧，我是来学操作系统的，您跟我说这干吗，还说了 12 个步骤，这与我何干？

前面所说的各种缓存，是为了引出 CPU 中的缓存。CPU 中的一级缓存 L1、二级缓存 L2，它们都是 SRAM，即静态随机访问存储器，它是最快的存储器啦。也许您又说：“是啊，这个我知道，那又怎么样？”

好啦不卖关子啦，也许您只听说 L1、L2、SRAM，但您可能不知道的是 SRAM 是用寄存器来存储数据的，这就是 SRAM 快的原因。而寄存器为什么快呢？原因是寄存器是使用触发器实现的，这也是一种存储电路，工作速度极快，是纳秒级别的，您想寄存器能不快吗，这和 CPU 的速度是一个级别啦。硬件我也不是很了解，所以只能跟您说到这了，点到为止，对于硬件背后的原理，咱们蜻蜓点水即可。

巧妇难为无米之炊，不管是指令，还是数据，CPU 内部总该有个地方存放它们，否则 CPU 这位巧妇连锅都没有，还怎么给大家烧一桌好菜呢？所以，寄存器是给 CPU 处理数据的场所。

到这里，大家心里应该对寄存器有个感性认识了，这样学起来，似乎看得见摸得着了。

CPU 中的寄存器大致上分为两大类。

- 一类是其内部使用的，对程序员不可见。“是否可见”不是说寄存器是否能看得见，是指程序员是否能使用。CPU 内部有其自己的运行机制，是按照某个预定框架进行的，为了 CPU 能够运行下去，必然会有一些寄存器来做数据的支撑，给 CPU 内部的数据提供存储空间。这一部分对外是不可见的，我们无法使用它们，比如全局描述符表寄存器 GDTR、中断描述符表寄存器 IDTR、局部描述符表寄存器 LDTR、任务寄存器 TR、控制寄存器 CR0～3、指令指针寄存器 IP、标志寄存器 flags、调试寄存器 DR0～7。
- 另一类是对程序员可见的寄存器。我们进行汇编语言程序设计时，能够直接操作的就是这些寄存器，如段寄存器、通用寄存器。

虽说第一类的程序是不可见寄存器，我们没办法直接使用，但它们中的一部分还得由咱们给初始化呢。

比如全局描述符表寄存器 GDTR，以后咱们还要通过 lgdt 指令为其指定全局描述符表的地址及偏移量。对于中断描述符表寄存器 IDTR，咱们也是要通过 lidt 指令为其指定中断描述符表的地址。而局部描述符表寄存器 LDTR，可以用 lgdt 指令为其指定局部描述符表 ldt（但我们效仿了现代操作系统，未用局部描述符表 ldt）。对于任务寄存器 TR，我们也要用 ltr 指令为其指定一个任务状态段 tss。对于 flags 寄存器，我们也有办法设置它，系统提供了 pushf 和 popf 指令，分别用于将 flags 寄存器的内容压入栈，将栈中内容弹到 flags 寄存器。额外说一句，ldt 和 tss 都位于 gdt 中。这些方面的内容咱们在学习保护模式后都会讲到。

在实模式下，默认用到的寄存器都是 16 位宽的，咱们说说具体的寄存器吧。

段寄存器是做啥的呢？说到段寄存器存在的理由，得先说到内存访问机制。CPU 是用“段基址：段内偏移地址”的形式来访问内存的，如果您对此访存形式不了解，前面第 0 章中有解释过分段，而且在下一节中也是讲内存分段的理由。您合理安排阅读。

现在假设您已经了解分段机制啦，上面提到的“段基址：段内偏移地址”中的段基址，就是用段寄存器来存储的，段寄存器的作用就是指定一片内存的起始地址，故也称为段基址寄存器。尽管段基址在实模式下要乘以 16，在保护模式下只是个选择子（保护模式中会讲），但其作用就是指定一片内存的起始地址。而段内偏移地址，顾名思义，仅仅相对于此起始地址的偏移量。

访问内存，是要通过地址总线，给地址总线一个数字（也就是地址），地址总线就能找到以该数字为地址的内存。可是这个数字是哪来的呢？对于首次访问内存之前，其内存地址肯定是要放在与内存不同的存储介质中更合适，也更容易。如果用内存来存储内存地址，首先访问该内存就是个问题，该内存的地址是什么？这就跟先有鸡还是先有蛋的本源问题一样了。您可能会有疑问，地址也只是个数字，把数字存放在内存中有什么不行的？我这里所说的并不是内存寻址，您说的这个数字只是该内存单元中的内容，这是内存寻址，前提是您已经给地址总线提交了保存该数字的内存地址。我说的是提交给地址总线的地址是从哪里获得的。不知道我说清楚了没有，再举个例子，我想用木材做一个船模，我不可能还用木质工具去加工它，只能用铁器等比木更硬的材料通过削、磨等方式将它加工出来。换在计算机中也一样，访问内存就要提供地址，初次访问内存时，该地址要么用立即数，要么存储在某个存储器中能让 CPU 取出来再访问内存，肯定不能用内存本身来存。由于寄存器比内存更高级，CPU 更能接受，所以就用寄存器来存储内存地址。由于要指定的是内存中的一段区域的起始地址，所以称之为段基址寄存器，也称段寄存器，无论是在实模式下，还是保护模式下，它们都是 16 位宽。

图 3-3 所示就是实模式下的段寄存器。

段寄存器(sreg)都是16位宽

15　　0	
CS	代码段寄存器
DS	数据段寄存器
ES	附加段寄存器
FS	附加段寄存器
GS	附加段寄存器
SS	栈段寄存器

▲图 3-3　实模式下的段寄存器

代码段简而言之就是把所有指令都连续排放在一起，形成了一个全部都是指令的区域，里面存储的是指令的操作码及寻址方式等。该区域可以在硬盘上的文件中，也可以是被加载后的内存中，总之是一段指令区域。它们内部都是紧凑挨着的，内容形式完全一样，只是存放的介质不一样而已。代码段寄存器 CS 就是用来指向内存中这段指令区域的起始地址。具体程序分段的解释，可见第 0 章。

数据段和代码段类似，只是这段区域中的内容不是指令，而是纯粹的数据，也就是说里面存储的是程序运行所需要的数据，属于指令的操作数。数据段寄存器 DS 便是用来指向此数据区域的起始地址。

栈段是在内存中，硬盘文件中可真没有。一般的栈段是由操作系统分配指定的，所以是属于被加载到内存后才有的。本章后面还会讲栈，这里大家就先当它是一段内存区域就好。栈段寄存器 SS 就是用来指向此区域的起始地址。

代码段、数据段、栈段寄存器从名字上就较容易理解，那三个附加段寄存器是干吗的？其实就是多给大家提供几个段寄存器用而已，多几个寄存器用不是更好吗，省得紧巴巴的，纯粹是为了方便大家。

值得说明的是在 16 位 CPU 中，只有一个附加段寄存器——ES。而 FS 和 GS 附加段寄存器是在 32 位 CPU 中增加的。我们使用的是 32 位 CPU，并不是说 32 位 CPU 在实模式下的 16 位环境中就不能用 FS 和 GS 寄存器。32 位的 CPU 兼容 16 位 CPU 的特性，就像一个小学生也可以穿中学生的衣服一样，无非是多了个可用的资源。

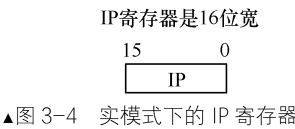
IP 寄存器是不可见寄存器，CS 寄存器是可见寄存器。这两个配合在一起后就是 CPU 的罗盘，它们是

给 CPU 导航用的。CPU 执行到何处，完成要听这两个寄存器的安排。为什么要用两个寄存器？因为指令是在内存中，访问内存就要用“段：段内偏移”的形式，所以 CS 寄存器用来存代码段段基址，IP 寄存器用来存储代码段段内偏移地址，同 CS 寄存器一样都是 16 位宽。

其实这两个寄存器没什么神奇的，并不是它们真地决定了 CPU 的航向，只是 CPU 的航向被存入了这两个寄存器之中。之前说过啦，指令在逻辑上是紧凑的，这样 CPU 便能连续不断地执行下去，天荒地老，直到断电。CPU 执行完一条指令后，顺便就把下一条指令的内存地址读进来。注意看，读入的是下一条指令的内存地址，这有两个关键字，一个是内存，另一个是地址。刚刚说过啦，是内存就应该用“段基址：段内偏移地址”的机制来访问，是地址就该有地方存放。您想，即使是洗菜，菜也得先找个盘子或盆之类的器具盛着再拿到水龙头下冲洗。在 x86 体系架构中，本着先满足“段基址：段内偏移”的形式，这个地址就分开存到了代码段 CS 寄存器和指令指针 IP 寄存器。在执行当前指令的同时，在不跨段的情况下，CPU 以“当前 IP 寄存器中的值+当前执行指令的机器码长度”的和作为新的代码段内偏移地址，将其存入 IP 寄存器，再到该新地址处读取指令并执行。如果下一条指令需要跨段访问，还要加载新的段基址到 CS 寄存器。此后，继续重复以上“取址、执行”的循环。

图 3-4 所示是实模式下的 IP 寄存器。

flags 寄存器是计算机的窗口，展示了 CPU 内部各项设置、指标。任何一个指令的执行、其执行过程的细节、对计算机造成了哪些影响，都在 flags 寄存器中通过一些标志位反映出来。有些指令需要满足某些条件才能执行，它们的条件是判断上一条指令的执行过程。所以标志寄存器中的标志位就成了这些指令所需要满足的条件。实模式下的 flags 寄存器是 16 位的，如图 3-5 所示。后面专门拿出一节讲 flags 寄存器，这里先一带而过。

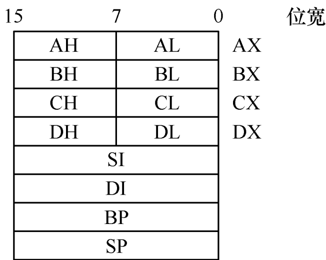


▲图 3-4 实模式下的 IP 寄存器



▲图 3-5 实模式下的 flags 寄存器

下面说说通用寄存器。无论是实模式，还是保护模式，通用寄存器有 8 个，分别是 AX、BX、CX、DX、SI、DI、BP、SP，它们的名称及关系如图 3-6 所示。



▲图 3-6 实模式下通用寄存器

拿 AX 寄存器举例，根据图 3-6 可知，AX 寄存器是由 AH 寄存器和 AL 寄存器组成的，它们都是 8 位寄存器，AX 寄存器的低 8 位，即 0～7 位，是 AL 寄存器。高 8 位，即 8～15 位，是 AH 寄存器。由于某种原因（如数学计算和 32 位保护模式等），16 位 AX 寄存器不够用了，将其扩展（Extend）为 32 位，在 AX 原有的基础上，增加 16 位作为高 16 位便是扩展的 AX，即 EAX。所以 EAX 归根结底也是由 AL、AH 组成的，AL 或 AH 值变了直接影响 EAX。这里提及了寄存器 eax 的原因是在实模式下虽然操作数是 16 位，但依然可以用 32 位寄存器，因为我们所讲的是 32 位 CPU 在实模式下的工作状态，并不是纯粹的 16 位 CPU，如 8086。

以上的这 8 个寄存器实际上是通用寄存器，通用是说每个寄存器的功能不单一，可以有多种用途，不像段寄存器 SS 那样只能用来放栈段基址，通用寄存器可以用来保存任何数据，包括地址（当然，地址也是一串数字，还是数据）。

虽说通用，但还是约定了它们的惯用法，除了通用的用途外每个寄存器肩负特定的功用。比如一般情况下，cx 寄存器用作循环的次数控制，bx 寄存器用于存储起始地址。这是大家约定俗成的东西，不这样做也可以，用其他通用寄存器也能完成任务。不过还是有个公共的约定更好，这样一些通用的函数，在为其传递参数时会方便很多。比如 BIOS 或 DOS 中断调用，一般情况下，cx 还就是用作循环次数的控制，有了这样公共的认知，函数在使用上会方便，且显得轻松很多。另外，一些指令已经固定用一些特定的寄存器作为参数了，比如 esi 寄存器作为很多有关数据复制指令的源地址，edi 作为目的地址。

简短介绍下各通用寄存器特定的功能，见表 3-2。

表 3-2　　　　　　　　　　　　　　　　　　　通用寄存器介绍

寄存器	助记名称	功能描述
ax	累加器（accumulator）	使用频度最高，常用于算术运算、逻辑运算、保存与外设输入输出的数据
bx	基址寄存器（base）	常用来存储内存地址，用此地址作为基址，用来遍历一片内存区域
cx	计数器（counter）	顾名思义，计数器的作用就是计数，所以常用于循环指令中的循环次数
dx	数据寄存器（data）	可用于存放数据，通常情况下只用于保存外设控制器的端口号地址
si	源变址寄存器（source index）	常用于字符串操作中的数据源地址，即被传送的数据在哪里。通常需要与其他指令配合使用，如批量数据传送指令族 movs[bwd]
di	目的变址寄存器（destination index）	和 si 一样，常用于字符串操作。但 di 是用于数据的目的地址，即数据被传送到哪里
sp	栈指针寄存器（stack pointer）	其段基址是 SS，用来指向栈顶。随着栈中数据的进出，push 和 pop 这两个对栈操作的指令会修改 sp 的值
bp	基址指针（base pointer）	访问栈有两种方式，一种是用 push 和 pop 指令操作栈，sp 指针的值会自动更新，但我们只能获取栈顶 sp 指针指向的数据。很多时候，我们需要读写在栈底和栈顶之间的数据，处理器为了让开发人员方便控制栈中数据，还提供了把栈当成数据段来访问的方式，即提供了寄存器 bp，所以 bp 默认的段寄存器就是 SS，可通过 SS：bp 的方式把栈当成普通的数据段来访问，只不过 bp 不像 sp 那样随 push、pop 自动改变

3.2.3　实模式下内存分段由来

CPU 中本来是没有实模式这一称呼的，是因为有了保护模式后，为了与老的模式区别开来，所以称老的模式为实模式。这情况就像所有同学坐在同一个教室里，本来没有老同学这一概念，但某天老师领着一个陌生人进入教室并和大家宣布：“这是新转到我们班的韩梅梅，大家欢迎新同学”。得，无形之中，大伙儿就成了老同学。

实模式的“实”体现在：程序中用到的地址都是真实的物理地址，“段基址：段内偏移”产生的逻辑地址就是物理地址，也就是程序员看到的完全是真实的内存。

不过要说实模式，咱们还得从 CPU 的发展说起，任何事物发展到今天，都有一段“合理”的过程，了解这一过程是怎么来的，有助于理解它今天的形态。

不知道各位同学当初学习汇编语言时有没有这样疑问：“老师都是拿 8086 型号的 CPU 举例，为什么不拿最新型号呢？用那么古老的 CPU 讲解知识，是否已经落伍太久了，我们学习的知识到社会上能用吗？”我记得当初学习汇编时，那时的 CPU 都是奔腾 2.8 了。我带着这样的疑问请教了老师，老师回答我说：“8086 是 Intel 历史上第一个 x86 的 CPU，也就是自那以后的 CPU 称为 286、386、486、586……即使是现在的奔腾也属于 x86 体系，道理是不变的，而且，用最简单的 8086 CPU 学习，这才更容易理解和看透 CPU 运行机制。”一番话彻底打消了我的疑虑，自那以后我才理解 x86 中的 x 原来是个变量，它指代 Intel 所有 86 系列的产品。

在 8086 之前的 CPU 是什么样呢？为什么 8086 就可以称为 CPU 界的里程碑呢？原因是这样的，在它之前的 CPU 前辈们，对内存的访问比 8086 还要“实诚”，它们没有段的概念，程序中要访问内存，需要把地址写死，也就是所谓的“硬编码”，这其实是很麻烦的，首先程序无法重定位，必须加载到内存中固定的位置，如果在此位置有其他程序在用，得，您先睡会，等它运行完成后我叫您。您看，得等人家运行完了腾出内存后才轮得到自己，可见程序对地址的依赖性之强。可用内存很多，但却因为某一个字节的内存被占用而让后来的程序等很久，这是很平常的事。有些开发人员等不及了，干脆把程序中的地址改成别的吧，重新编译后发现还是有某个地址被占用，还是没法上 CPU 运行，怎么办？再改地址……所以我估计那时的开发人员脾气都会很差，这人脾气差就容易伤肝，肝火一旺就会两鬓斑白，所以 IT 工程师还是很值得体恤的。

看着越来越多的程序员两鬓斑白，Intel 早期的工程师难以承受内心的自责，不顾自己的满头白发，熬了无数通宵之后，终于发明了“段”，即 CPU 访问内存用“段+偏移”的形式。这就是前面曾经讲解过访问内存用“段基址：段内偏移地址”的策略，它就是首次在 8086 上出现的。自那之后的 CPU 都是用这类思想访问内存，只是在形式上有小改动，难怪 8086 如此极负盛名了。为了支持段机制，CPU 中新增了段

寄存器，如 cs、ds、es 等。

8086 的地址总线是 20 位宽，也就是其寻址范围是 2 的 20 次方=1MB。但其内部寄存器都是 16 位的，若用单一寄存器来寻址的话，只能访问到 2 的 16 次方等于 64KB 的空间。

由于地址线位宽和寄存器位宽是没有必然联系的，所以大家不要觉得为什么寄存器不是 20 位的？这样通过寄存器寻址就能访问到 1MB 空间了，显得多“配套”。

例如 8086 的多种寻址方式中，有一种是基址寻址，这是用基址寄存器 bx 或 bp 来提供偏移地址的。例如 mov [bx], 0x5；这条指令便是将立即数 0x5 存入 ds：bx 指向的内存。

大家看，bx 寄存器是 16 位的，它最大只能表示 0xFFFF 的地址，也就是单一的一个寄存器无法表示 20 位的地址空间：1MB。也许有人会说，段基址和段内偏移地址都搞到最大，都为 0xFFFF。对不起，方案不成立，首先说会溢出，结果是 0xFFFE，地址不增反小了个 1。即使不溢出的话，其结果也只是由 16 位变成了 17 位，即两个 n 位的数字无论多大，其相加的结果也超不过 n+1 位，道理很简单，即使两个数都是 n 位能表示的最大数，两个相同的数相加，相当于乘以 2，也就是数值上等于左移一位而已。依然无法访问 20 位的地址空间。也许有人又有好建议了：CPU 的寻址方式又不是仅仅这一种，上面的限制是因为寄存器是 16 位，只要不全部通过寄存器寻址不就行了吗？段寄存器也是寄存器，同样也是 16 位，既然它必须得用，那就在偏移地址上下功夫，不要把偏移地址写在寄存器里了，把它直接写成 20 位立即数不就行啦？例如 mov ax, [0x12345]，这样最终的地址是 ds+0x12345，肯定是 20 位，解决啦。不错，这种是直接寻址方式，至少道理上讲得通，这是通过编程技巧来突破这一瓶颈，能想到这一点我觉得非常 nice。但是作为一个严谨的 CPU，既然宣称支持通过寄存器来寻址，那就要能够自圆其说才行，不能靠程序员的软实力来克服 CPU 自身的缺陷。于是，一个大胆的想法出现了。

为了让 16 位的寄存器寻址能够访问 20 位的地址空间（注意，我这里说的是通过寄存器寻址，因为只有通过 16 位的寄存器去寻址才会受到 16 位的限制），CPU 工程师定位到根本瓶颈是在段寄存器，它要是能提供 20 位的段地址，哪怕偏移地址是 1 也照样可以访问到内存的各个角落。于是，通过先把 16 位的段基址左移 4 位后变成 20 位，再加段内偏移地址，这样便形成了 20 位地址，只要保证了段基址是 20 位的，偏移地址是多少位都不关心了，从而突破了 16 位寄存器作为偏移地址而无法访问 1MB 空间的限制。

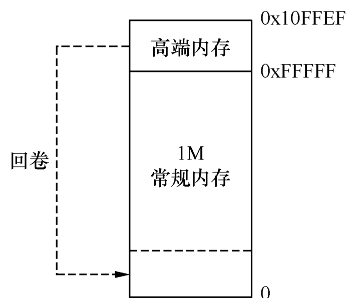
有了 20 位地址便能访问到 20 位的空间，虽然解决了一个大问题，但是引入了一个小问题。

还拿 0xFFFF 来说，现在能访问的最大的地址是 0xFFFF：0xFFFF，经过左移段基址 4 位后得到的最大地址是：0xFFFF*16+0xFFFF=0xFFFF0+0xFFFF

=0xFFFFF+0xFFF0=1M+16*4KB-16-1=0x10FFEF。这公式有点晕是吗？其实最后结果 0x10FFEF 是最重要的，前面的推算就是想告诉大家，按照新方法获取地址，可以得到的最大地址是 1MB+64KB-16 字节，因为这是空间范围，所以要减去 1 得到地址范围。

大家看到了，当初费了好大周折才搞定了能够访问 20 位地址空间，现在反而有点过了，过头的原因是段基址为 0xFFFF0，偏移地址应该小于等于 F 就对啦，而这个偏移地址却是 0xFFFF，超出了 0xFFF0 的空间，也就是多出来 64K-16 字节，这部分内存就是传说中的高端内存区（High Memory Area，HMA）。可是这部分内存不存在，怎么处理呢？

答案说出来吓你一跳：不用处理。您想，8086 一共就 20 条地址，地址线是从 0 开始的，即 A0～A19，所以其地址空间才是 1MB 的啊。内存地址 0xFFFFF+是要用到 A20 地址线，可是 8086 它没有啊，只能接收 20 位长的地址。所以由于超过了 20 位而产生的进位，就给丢掉了。其作用相当于把地址对 1MB 取模了。举例，如 0xFFFFF+2，理论上是变成了 0x100001。但由于只能容纳 20 位长的数据，所以最终结果是 0x00001。这是地址回卷的效果，即超过最大范围后，从 0 重新开始计数。回卷英文称为 wrap-around，示意图如图 3-7 所示。



▲图 3-7　实模式下高端内存回卷

这就引出了从实模式到保护模式要打开 A20 地址线的问题，不过这部分在讲保护模式时咱们再说。

3.2.4　实模式下 CPU 内存寻址方式

上一节跟大家讲述了“段基址+段内偏移地址”的由来，在实模式下 CPU 访问数据将按照此方法来进行。没有规矩不成方圆，CPU 访问数据也得有个章法可循。其实一切所谓的格式都是一种“协议”、“约定”，“约定”的原因是为了省事，让大家共同按照某种约定行事，这样服务的提供方就不必为满足各种各样需求方而煞费苦心。为了 CPU 设计更容易，CPU 访问数据的形式也需要提前“约定”好，这就是所谓的寻址方式。下面把 8086 的寻址模式和大家说说。

寻址方式，从大方向来看可以分为三大类：

（1）寄存器寻址；

（2）立即数寻址；

（3）内存寻址。

在第三种内存寻址中又分为：

（1）直接寻址；

（2）基址寻址；

（3）变址寻址；

（4）基址变址寻址。

要讲寻址方式，得先知道什么是寻址。

从名字上看，寻址就是寻找地址，寻找谁的地址？CPU 眼里只有二进制数，所以这是 CPU 在寻找“数”的地址。这个“数”可以源操作数，也可以是目的操作数（顺便提一句，Intel 汇编语言语法是“指令目的操作数，源操作数”），简而言之，寻址就是找到“数”的所在地，从哪来，往哪去。

- 寄存器寻址

最直接的寻址方式就是寄存器寻址，它是指“数”在寄存器中，直接从寄存器中拿数据就行了。例如下面用 mul 指令实现 0x10*0x9。

```
mov ax, 0x10
mov dx, 0x9
mul dx
```

以上三条指令都是寄存器寻址。

第一条命令是将 0x10 存入 ax 寄存器，第二条命令是将 0x9 存入 dx，第三条指令是求 ax 和 dx 的乘积，乘积的高 16 位在 dx 寄存器，低 16 位在 ax 寄存器。只要牵扯到寄存器的操作，无论其是源操作数，还是目的操作数，都是寄存器寻址。上面的第一、二条指令，它们的源操作数都是立即数，所以也属于立即数寻址。

- 立即数寻址

什么是立即数？立即数就是常数。常数就常数呗，为什么拐着弯叫别的名呢？

是这样的，指令由操作码和操作数组成，得到一个数往往不容易，或者说不那么直接。这个数要么在寄存器中，要么在内存中，都是间接给出的，所以得到数就要花费一些 CPU 周期。如果操作数“直接”存在指令中，直接拿过来，立即就能用了。为了突显“立即就能用”的高效率，此数便称为立即数。立即数免去了找数的过程，CPU 最喜欢它啦。如：

```
mov ax,0x18
mov ds, ax
```

第一条指令中的源操作数 0x18 是立即数，目的操作数 ax 是寄存器，所以它既是立即数寻址，也是寄存器寻址。第二条指令中，源操作数和目的操作数都是寄存器，所以纯粹是寄存器寻址。

提醒一下，像这样的寻址也是立即数寻址：

```
mov ax, macro_selector
mov ax, label_start
```

第一条指令的源操作数 macro_selector 是个宏，第二条指令的源操作数 label_start 是个标号，这两个

在编译阶段会转换为数字，最终可执行文件中的依然是立即数。

- 内存寻址

以上两种寻址方式，操作数一个是在寄存器中，一个是在指令中直接给出。它们都不在内存中。操作数在内存中的寻址方式称为内存寻址。

CPU 中有很多寄存器，有些是程序员不可见的，它们是为了 CPU 正常运行而存在的，属于 CPU 运行框架内的需求。CPU 给程序员用的寄存器并不是很多，所以操作数一多起来的时候，基本就倒腾不开了。内存空间相对就大多了，于是 CPU 工程师们自然而然想到了用内存来存储操作数。另外，用立即数寻址，得提前知道立即数是多少，否则还真用不了。而且，大多数时候操作数位于内存中的某个位置，只知道操作数所在的内存地址，不知道操作数的值，更谈不上将其变成立即数用在指令中了，这就更加有理由让内存寻址成为“应该”。

由于访问内存是用“段基址：偏内偏移地址”的形式，特别强调一下，此形式只用在内存访问中。默认情况下数据段寄存器是 DS，即段基址已经有了，只要再给出段内偏移地址就可以访问内存了，最终起决定作用的、有效的是段内偏移地址，所以段内偏移地址称为有效地址。

以下所说的寻址方法都是在内存中寻址的方法。

- 直接寻址

直接寻址，就是将直接在操作数中给出的数字作为内存地址，通过中括号的形式告诉 CPU，取此地址中的值作为操作数。如：

```
mov ax, [0x1234]
mov ax, [fs:0x5678]
```

0x1234 是段内偏移地址，默认的段地址是 DS。这条指令是将内存地址 DS：0x1234 处的值写入 ax 寄存器。还记得规则吗？段基址*16 变成 20 位地址后，再加上段内偏移地址 0x1234，当然结果还是 20 位地址。

第二条指令中，由于使用了段跨越前缀 fs，0x5678 的段基址则变成了 gs 寄存器。最终的内存地址是 gs 寄存器的值*16+0x5678，CPU 到此内存地址取值再存入 ax 寄存器。

注意啦，不要和立即数寻址混了，立即数寻址中的数字是直接拿来就用作操作数了，直接寻址中的数字是用来进一步寻址的。

- 基址寻址

基址寻址，就是在操作数中用 bx 寄存器或寄存器作为地址的起始，地址的变化以它为基础。注意看啦，这里说的是只能用 bx 或 bp 作为基址寄存器。用寄存器作为内存寻址，在实模式下就是这样的，必须用 bx 或 bp 寄存器。到了保护模式下就没这个限制了，基址寄存器可选择的很多，可以是全部的通用寄存器。

说明一下，bx 寄存器的默认段寄存器是 DS，而 bp 寄存器的默认段寄存器是 SS，即 bp 和 sp 都是栈的有效地址。如果你忘记了什么是有效地址，第 0 章有介绍，简而言之有效地址就是指偏移地址。例如“add word[bx]，　0x1234”这条指令将 0x1234 加上内存地址 ds：bx 处的值后再存入内存地址 ds：bx 中。这条指令用到了立即数寻址和内存基址寻址两种方式。同样，ds 也要乘以 16 后再加上 bx 寄存器的值，这是实模式下恒久不变的步骤。

再看个用 bp 寄存器作为基址寄存器的例子。前面说过啦，用 bp 寄存器作为偏移地址时，其默认的段寄存器是 SS。这就是说，bp 是用来访问栈的。为什么已经有了 sp 寄存器来“专门”访问栈，还要再单独准备个 bp 呢？sp 寄存器作为栈顶指针，相当于栈中数据的游标，这是专门给 push 指令和 pop 指令做导航用的寄存器，push 指令往哪个内存压入数据，popd 将哪个地址的数据弹出栈，都要看 sp 的值是多少。在实模式下，CPU 字长是 16，所以实模式下的 push 指令默认情况下是压入 2 字节的数据，其工作原理可以分为两步，假如执行 push ax：

```
1 sub sp，2            先将 sp 的值减去
2 mov sp，ax           再将 ax 的值 mov 加到新的 sp 指向的内存
```

实模式下 pop 指令，其工作原理也分为两步，假如执行 pop ax：

```
1 mov ax， [sp]        先将 sp 指向的值 mov 到 ax
2 add sp，2            再将 sp 的指针+2
```

这么重要的指针，可不是随便敢动的，如果动错了，栈中数据错位了，对整个程序的运行将造成不可估量的灾难。

栈也是内存中的区域，既然 sp 不敢乱动的话，该如何访问到此区域中的数据呢？

有需求的地方就有解决的办法，得看为什么要访问栈。

访问栈有两种方式，一种是把栈当成“栈”来使用，也就是用 push 和 pop 指令操作栈，但这样我们只能访问到栈顶，即 sp 指向的地址，没有办法直接访问到栈底和栈顶之间的数据。很多时候，我们需要读写栈中的数据，即需要把栈当成普通数据段那样访问。举个需要直接写栈的例子，比如标志寄存器 eflags 无法直接修改，只能用 pushf 指令把 eflags 寄存器的内容压到栈中，我们在栈中修改完后，再用 popf 把它弹回到 eflags 中。处理器为了让开发人员方便控制栈中数据，提供了这种把栈当成数据段来访问的方式，可以用寄存器 bp 来给出栈中偏移量，所以 bp 默认的段寄存器就是 SS，这样就可通过 SS：bp 的方式把栈当成普通的数据段来访问了。

再举个需要直接读栈中数据的例子。一个最重要的应用就是在栈中保存局部变量和函数参数。这里虽然是想解释 bp 寄存器的应用，但若不拿例子解释的话会显得很刻意，最好的例子是用 32 位环境下的 ebp 寄存器来解释，ebp 和 bp 的应用原理是一样的。在 32 位环境下，ebp 就应用在堆栈框架中，堆栈框架是编译器在栈中为局部变量分配内存空间的方式，局部变量存在于函数中，因此有关堆栈框架的汇编指令是在函数的开头和结尾处。下面通过这段代码了解堆栈框架的原理。

```
1 int a = 0;
2 function(int b, int c) {
3  int d;
4 }
5 a++;
```

假设这是在 32 位下。

（1）调用 function(1,2)；按照 C 语言调用规范，参数入栈的顺序从右到左：会先压入 2，再压入 1。每个参数在栈中各占 4 字节。

（2）栈中再压入 function 的返回地址，此时栈顶的值是执行“a++”相关指令的地址。

下面是堆栈框架的指令。

（3）push ebp；将 ebp 压入栈，栈中备份 ebp 的值，占用 4 字节。

（4）movebp, esp；将 esp 的值复制到 ebp，ebp 作为堆栈框架的基址，可用于对栈中的局部变量和其他参数寻址。

此时的 ebp 便是栈中局部变量的分界线。在这之后，esp 将自减一定的值为局部变量在栈中分配空间，该值取决于所有局部变量空间大小的总和。

（5）sub esp,4；由于函数 function 中有局部变量 d，局部变量是在栈中存放的，故 esp 要预留出 4 字节，专门给变量 d 使用。

终于到了应用 ebp 指针的时候，以 ebp 为基址对栈中数据寻址。

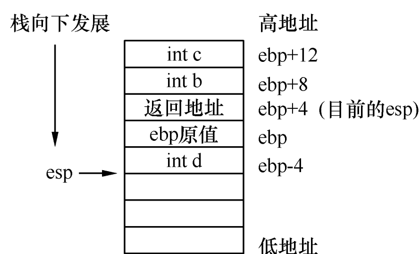
[ebp-4]是局部变量 d，对应上面的第（5）步。

[ebp]是 ebp 在栈中的备份，对应上面的第（3）步。

[ebp+4]是函数的返回地址，对应上面的第（2）步。

函数中的参数 b 是用[ebp+8]访问，参数 c 用[ebp+12]访问，对应上面的第（1）步。

栈中数据的布局如图 3-8 所示。



▲图 3-8　堆栈框架栈中布局示意图

（6）函数结束后跳过局部变量的空间：mov esp, ebp。

（7）恢复 ebp 的值：pop ebp。

至此函数中堆栈框架的指令结束了，然后是返回指令 ret，接着主调函数中执行“add esp,8”来回收参数 b 和 c 的空间。

例子说完了，您看，用[e]bp 寄存器作为栈的指针来访问栈中数据还是非常方便的。

另外说一下，堆栈框架的工作是为函数分配局部变量空间，因此应该在刚刚进入函数时就进行为局部变量分

配空间的工作，离开函数时再回收局部变量的空间，所以堆栈框架的创建和回收工作分别是在进入函数和离开函数时进行的。为了在名称上突显堆栈框架这两个阶段，有一条指令叫 enter，它是在进入函数时执行的，其功能就是备份 ebp 并使 ebp 更新为 esp，即先“push ebp”再“mov ebp，esp”，因此第 3～4 步的两条指令通常会由一条 enter 指令来代替。另一条指令是 leave，它是在离开函数时执行的，其功能是回收局部变量的空间并恢复 ebp 的值，即先“mov esp, ebp”再“pop ebp”，因此第 6～7 步的两条指令也通常由一条 leave 指令来代替。

- 变址寻址

变址寻址其实和基址寻址类似，只是寄存器由 bx、bp 换成了 si 和 di。si 是指源索引寄存器（source index），di 是指目的索引寄存器（destination index）。两个寄存器的默认段寄存器也是 ds。

```
mov [di]，ax               ;将寄存器 ax 的值存入 ds：di 指向的内存
mov [si+0x1234], ax        ;变址中也可以加个偏移量
```

变址寻址主要是用于字符搬运方面的指令，这两个寄存器在很多指令中都要成对使用，如 movsb，movsw，movsd 等，我们的代码中也用到了此命令，讲到时会细说，目前大家有兴趣可自行查阅。

其实单纯的变址寻址没什么新鲜的，它只是为了配合基址寻址，用来实现基址变址寻址。

- 基址变址寻址

从名字上看，这是基址寻址和变址寻址的结合，即基址寄存器 bx 或 bp 加一个变址寄存器 si 或 di。如：

```
mov [bx+di], ax
add [bx+si], ax
```

第一条指令是将 ax 中的值送入以 ds 为段基址，bx+di 为偏移地址的内存。第二条指令是将 ax 与[ds：bx+si]处的值相加后存入内存[ds：bx+si]。

在咱们项目里的汇编代码里还没有这么用的，不做过多演示，大家知道有这种形式就行。

CPU 访问数据的方式看上去很死板（有些寄存器是规定的），原因是一种寻址方式对应一种电路实现，增加一种寻址方式，会增加硬件电路的复杂性，所以寻址方式是有限的。

就拿“mov ax，[bx+si]”来说，您有没有疑问，换成 mov ax，[cx+dx]行不行？这在逻辑上没有任何问题，我不觉得它们之间有什么不同，咱们关心的是 bx+si 要等于 cx+dx 就行了。这是人类的理解，我们不知不觉中站在了抽象层来看待这个偏移地址，即咱们关注的是这个数是否正确，而不管这个数是怎么来的。

对于有效地址（段内偏移地址），不管其形式是寄存器、立即数，还是内存中的值，甚至是个表达式，它在人类眼里只是个数字，按理来说都是一样的，不应该强调具体形式。可是这对计算机硬件实现来说却是截然不同的，所以才又细分了这么多的寻址形式。给大家举个生活中的例子。

大家吃饭时对于主食的选择，有人喜欢吃饼，有人喜欢吃米饭。虽然它们都是主食，但这两种食物的加工方法必须是不同的，饼是用饼铛做出来的，米饭是用锅蒸出来的。想吃饼类食物，其制作过程就一定要用到饼铛，这是最底层不变的东西。无论做出的是鸡蛋饼，还是馅饼，其上层形式无论多丰富，万变不离其宗，都是经过饼铛烹制而成的。总结出：一类食物种类对应一类工具，想创造一种新食类，就要发明新的工具（当然我这么总结是不严谨的，用一口锅也能做出很多种不同的食物，这确实是可能的）。结合 CPU 寻址方式，大家可以这么想，一种寻址方式相当于一类食物，而其相应的硬件电路是某种制造工具，上层形式万千的变化，归根结底就是这么几类硬件电路，而它是有限的。CPU 的寻址方式看似不灵活，有的甚至要背下来，这是因为这和人类理解的方式是不一样的，人类是站在抽象层来看待寻址的，如果站在底层来看它们，就会理解在这么多的寻址方法中，为什么看似一样却又有这么多形式，就是因为 8086 在寻址方面的硬件电路做得简单有限，为了更简单，某些功能中使用的寄存器甚至要“写死”。如该用基址寻址时，电路中就只针对 bx 或 bp 寄存器，从硬件上就没考虑其他寄存器。

寻址方式到这里就讲完了。

咱们在实模式下的代码中还用到了 call、ret 和 jmp，下面说说它们的用法。

3.2.5　栈到底是什么玩意儿

CPU 中有栈段 SS 寄存器和栈指针 SP 寄存器，它们是用来指定当前使用的栈的物理地址。换句话说，

要想让 CPU 运行，必须得有栈。栈是什么?干吗用的？本节将给大家一个交代。

还记得数据结构中的栈吗？那是逻辑上的数据存取结构，是种如何用这种数据结构来存取数据的描述。

在用户进程空间中，堆是堆，栈是栈，但堆栈却是人们常说的栈，和堆没关系，所以，咱们后面为避免混淆，只说栈。

栈是线性表的一种，什么是线性表？如果提出这样的问题，我想您可能不清楚什么是线性。线性就是具有线的性质，就像一条线一样，连续性强，从一个方向到另一个方向。线上没有面积的概念，不管是直线，还是曲线，在线上任意位置只能容纳一个数据对象。线性表简而言之就是一个线性存储单元，结构中每个元素都有一个前驱和一个后继元素，且仅各有一个。这就是线性的体现：连续，且任意位置只有一个元素。栈也是这样，不过不同的是数据的存取都在一端进行，这一端称为栈顶，另一端作为存储单元的基址永远不动，称为栈底。这就是上学时，老师们常常说的后进先出，先放进去的数据要在最后才能取出，后放进去的数据最先被取出。

这里我就不用举汉诺塔这样经典的例子了，毕竟上学时都听得太多了。我说个大家都认同的事实：大家肯定挤过公交车吧，尤其是早班车和末班车，车厢里人挤人，站都站不稳。先挤上车的乘客其实很倒霉的（有座儿不算），因为他要在最后下车，在拥挤的车厢中折磨的时间最长。后挤上车的乘客在下车的时候还是很高兴的，因为他会是第一个下车。所以，挤公交车就是典型的后进先出。车厢就相当于栈，乘客就相当于栈中存取的元素，这个例子其实还算生动。

举的例子虽然很常见，但这对于已经理解栈的人来说，我像是在说废话一样没新意。对于不理解栈的人来说，可能是依然像说废话一样，说了也意会不到栈是什么。我非常理解这种心情，记得当初我在学网络时，老师说只要在路由器上把三层（网络层）IP 协议（不是指令指针寄存器 IP）禁用，四层（传输层）上的 TCP 或 UDP 协议自然就不可用了。老师为了让我们明白这种依赖关系，甚至不惜举出这样的例子：如果不想让某人说话，最简单的办法就是给让其睡着，而不是劝他保持安静。这个例子非常浅显易懂，但用例子来理解理论知识，依然让我有点摸不着头脑，这可不是比喻恰不恰当的事，知识是严谨的，不是比喻出来的。如果您现在也有这样的体会，没关系，以后会不断接触栈的，熟了自然就理解了，这只是时间问题。

初次学习数据结构时，不容易理解其本质，我当初在学习这门课时，感觉云里雾里的，似乎明白似乎又不懂，老师让不懂的同学提问，我又不知道该怎样描述问题，不知道哪里不懂。同样的定义，同样的文字描述，每个人理解的都不一样。就像鱼和小鸟，鱼认为自己离开水就会死，水就是生命，小鸟也认为没水会渴死，水也是生命。但鱼和小鸟对水的理解能一样吗？

赶紧回来，还是说咱们的正事。栈只是一种抽象概念，是一种虚拟出来的数据存取方法。其实现形式是不限的，只要满足栈的定义就可以。

- 首先得是线性结构，并且数据的存取在线性结构的一端进行。如果您愿意，可以用链表来实现，也可以用数组来实现，它们都是线性数据结构。
- 其次需要维护一个指针，用它来指向线性结构的一端，数据存取都通过此指针。

前面又比喻又回忆的，说了这么多，栈能够干什么呢？

栈是一种很伟大的发明，可以解决很多难题。

- 表达式计算，如中缀表达式和后缀表达式的转换。
- 函数调用，无论是嵌套调用或递归调用，用来维护返回地址。
- 深度优先搜索算法。

到目前为止，我们说的只是数据结构中的栈，这是逻辑上的，最终我想表达的是内存中的栈，这是物理上的。把数据结构中的栈的概念用物理硬件来实现，这就是我们要说的栈。它同数据段、代码段一样，是个内存中的区域，也就是栈段寄存器 SS 和栈指针 SP 所指向的内存区域。我们常听说的栈溢出，指的就是这个内存区域无法容纳数据了。

硬件是如何实现这个栈的呢？

还是那句话，首先得满足栈的概念，具备栈的特性，即使是硬件也不能例外，必须满足上面提到的这两个条件：一个是线性结构，另一个是在栈顶对数据存取。因为它毕竟造的是栈，不具备这些就不叫栈了。

线性结构这个简单，内存就是，直接用物理内存存取最方便了，咱们要做的就是给栈指定一片内存区域，区域的起始地址作为栈基址，存入栈基址寄存器 SS 中，另一端是动态变化的，用栈指针寄存器 SP 来指定。栈在使用过程中是向下扩展的，所以栈顶地址肯定小于栈底地址。

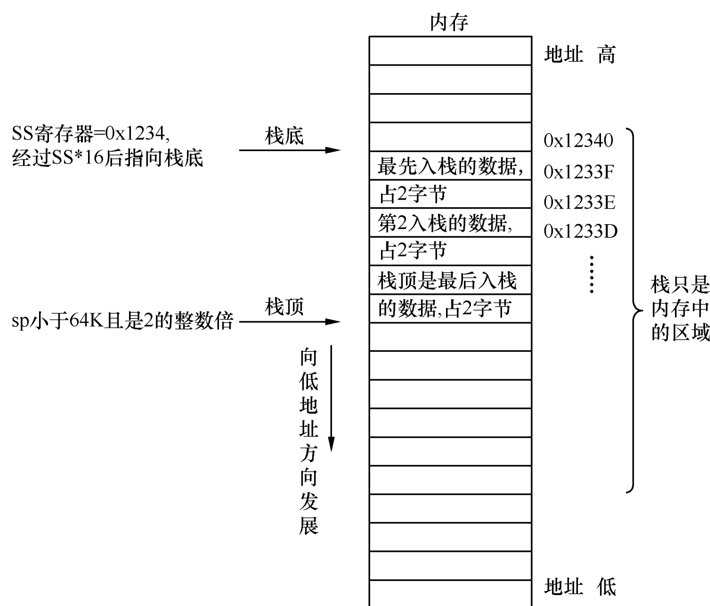
栈既然是一片内存区域，访问内存就要用“段基址：段内偏移地址”的形式，所以栈中的内存地址也是用“段基址 SS 的值*16+栈指针 SP（段内偏移地址）形成的 20 位地址”访问到的。

由于是硬件实现的栈，故硬件提供了相应的方法来存取栈，即 push 和 pop 指令。push 指令负责把数据压入栈，pop 指令功能相反，将其从栈中取出。不过我刚才说的不全面，栈的出口和入口都是栈顶，push 把数据压向哪里，它得知道栈顶在哪里才行。pop 指令也一样，它得知道哪里是栈顶才能从栈中取出正确的数据。这正是栈指针寄存器 SP 的作用，此寄存器中的值是段内偏移地址，是栈顶相对于栈底的偏移量。

栈顶（SP 指针）是栈的出口和入口，它指向的内存中存储的始终是最新的数据。push 和 pop 就是操作这个指针所指向的内存。由于栈从高地址向低地址发展，所以栈顶、栈指针指向的地址会越来越低。push 压入数据的过程是：先将 SP 减去字长，目的是避免将栈顶的数据破坏，所得的差再存入 SP，栈顶在此被更新，这样栈顶就指向了栈中下一个存储单元的位置。再将数据压入 SP（新的栈顶）指向的新的内存地址。pop 指令相反，既然是在栈中弹出数据，栈指针寄存器 SP 的值应该是增大一个数据单位。由于要弹出的数据就在当前栈顶，所以在弹出数据后，才将 SP 加上字长，所得的和再存入 SP，从而更新了栈顶。这样 SP 就指向了上一个存储单元的位置。

上面提到的字长，是指 CPU 的字长，即一次可处理的数据的长度。在实模式下的字长是 16。

物理内存中的栈如图 3-9 所示。



▲图 3-9　栈在内存中的示意图

注意啦，如图 3-9 所示，虽然栈是向下发展的，但栈也是内存，访问内存依然是从低地址往高地址，假如当前栈顶是 0x1233E，栈顶数据占 2 字节的话，其范围是 0x1233E～0x1233F。

个人觉得，这个硬件中的栈让人感到神秘，主要有两方面原因。

一方面是栈指针不是自己维护，这不像咱们在高级语言中自己创建的栈那样，指针的一举一动都是自己在操作。不直接受控的东西往往让人心存忧虑和有点小恐慌。其实即使是这里的硬件栈，咱们也可以自己维护指针，如 push ax 可以这样代替：

```
mov bp,sp
sub bp, 2
mov [bp],ax
```

bp 默认的段寄存器就是 SS，用 bp 的时候直接操作的便是栈。bp 就相当于栈指针啦，自己维护毕竟太麻烦，有直接省事的干吗不用呢。

另一方面，栈就是一片内存区域，只不过“经常”操作这片内存的指令不是 mov，而是 push、pop，这两条指令无非是自动维护存取数据的位置（SP 寄存器的值）而已，大家用 mov 来操作这片内存，不是也得要给出存取地址吗。这样看来，它和普通的数据段没什么不同，不要觉得它比金字塔还神秘。

一定要注意，push 和 pop 操作是要成对出现的，这样才能维护栈平衡。否则，光 push，不 pop，有进没出，这栈很快就溢出啦。切记，一个 push 要对应一个 pop，每键入一个 pop 指令，一定要清楚它对应的是哪个 push。

好啦，栈就先说这么多，不摸索实际东西的话还是不能真正掌握和理解，本书强调实践，纸上谈兵可来不了真知识。

3.2.6　实模式下的 ret

由于指令都是存在内存中，所以 CPU 也要访问内存才能拿到要执行的指令。既然是访问内存，就免不了要用“段基址：段内偏移地址”的形式，对于指令来说，CS 寄存器是代码段段基址，IP 寄存器中的是代码段的段内偏移地址。经过寄存器 CS*16 后再加上 IP 寄存器的值，所得的和就是指令存放的内存地址。CPU 在此内存地址处获得指令并执行。所以说，CPU 前进的方向永远是 CS：IP 这两个寄存器。

由于不能直接往 IP 寄存器中赋值（如同 mov ip ， xxxx 这样的指令是错误的），即使能的话，太灵活了反而更麻烦，很多相关数据结构都要自己设置。例如 call 指令要压入返回地址，进入中断时，要压入代码段寄存器 CS、指令指针寄存器 IP、状态寄存器 flags，除此之外还要考虑特权级，若特权级有变化，还要压入栈段寄存器 SS 及栈顶指针寄存器 SP。所以为了大家使用方便，CPU 中提供了很多可以改变 CPU 执行流的指令，它们在内部实现上，包含了许多微操作，用于设置相关数据结构。人家已经将这些步骤封装成一个指令，咱们直接调用就好。所以其表面上看来是一个指令，实际上其内部干的活可不少呢。例如即将讲述的 call、ret、jmp 都是此类指令，它们在原理上是修改寄存器 CS 和 IP 的值，将 CPU 导向新的位置。

call 指令用来执行一段新的代码，让 CPU 踏上新的征途，为避免这是一条不归路，还需要返回指令 ret 来帮忙。

生活中，我们去某个地方办事、游玩，虽然表面上看一下子就直奔主题，乘坐各种交通工具直接过去了，但其实在大家的潜意识中，去某个地方之前，是先考虑了怎样才能回来，去之前就已经想好了回来的路，如果您没有考虑如何回来的话，您这份不羁的洒脱还是很值得我向往的。

所以，我觉得在讲述 call 之前，大家最好要了解下 ret。

call 指令调用一个函数时，发生了什么呢？压入返回地址，为将来能够回来埋下伏笔。call 指令不负责“回来”，它只负责如何“去”，回来的工作要交给 ret。

call 指令不是一去不回头，它执行完目标函数后还是要回来的，所以它得提前把回来的路（返回地址）记好了，对于 CPU 来说，它是靠程序计数器 PC 来指路的，所以路就在 PC 中。凡是调用 call 指令，CPU 就要找地方把返回地址存起来以备将来函数执行时有路可以返回。在哪里保存返回地址合适呢，这需要考虑函数嵌套调用的问题。由于函数有可能嵌套调用，也就是随着函数调用的层数增加，会有更多的返回地址需要保存。用宝贵且有限的寄存器来保存无数的返回地址，这显然是不现实的。内存空间相对是无限的，在内存中保存数量未知的返回地址比较理想。栈这种数据结构是再适合不过的，利用其后进先出的特性，可以保证函数嵌套调用及嵌套返回顺序的一致性，而且其空间只受限于内存大小。于是在内存中创建这样一个数据结构就完美地解决了这个问题，所以 CPU 在栈中保留程序计数器 PC 的值。在 x86 中的程序计数器是 CS：IP，具体保留 IP 部分还是 CS 和 IP 都保留，是要看目标函数的段基址是否和当前段基址一致，也就是说，是否跨段访问了。

保留的这个返回地址并不是给 call 指令用的，call 指令不会自动回来，它只会留下返回地址后并踏上新的征程，返回地址是给 ret 或 retf 指令准备的，也就是说，在目标函数中必须有这两个指令之一，CPU 才能回来。

ret（return）指令的功能是在栈顶（寄存器 ss：sp 所指向的地址）弹出 2 字节的内容来替换 IP 寄存器，注意，我这里说的是“内容”，ret 指令不管里面的内容是不是地址，它只负责把当前栈顶处的内容弹出栈

并用它为 IP 寄存器赋值。至于内容的正确性应该由程序员自己控制。ret 只置换了 IP 寄存器，也就是说，不用换段基址，属于近返回。既然我们称之为弹出栈，也就是说 ret 指令也要负责维护栈顶指针，由于栈是从高地址往低地址发展，所以被回收的栈顶空间应该是使 sp 指针值变大，故 ret 指令会使 sp 指针+2。

retf（return far）是从栈顶取得 4 字节，栈顶处的 2 字节用来替换 IP 寄存器，另外的 2 字节用来替换 CS 寄存器。同样，retf 也不会去检查从栈顶往上的 4 字节内容是不是偏移地址和段基址，它只负责弹出它们，并将它们分别载入代码段寄存器 CS 和指令指针寄存器 IP，由程序员负责栈中数据的正确性。换句话说，在用 ret 或 retf 之前，程序员应该知道此时栈顶中的数据是什么，能不能用作返回地址。段寄存器都换了，说明这属于远返回。retf 指令也要负责维护栈顶指针，所以 retf 指令会使 sp 指针+4。

小结一下，ret 和 retf 的区别便是 ret 用于近返回，retf 用于远返回。

ret 和 call 指令是需要配合使用的，注意了，是 ret 要以 call 为主，根据 call 的种类来选择 ret 或 retf。call 的种类从大方向上分也就两种，一是近调用，另一个是远调用。

call 和 ret 是一对配合，用于近调用和近返回，call far 和 retf 是一对配合，用于远调用和远返回。故：

如果 call 是近调用，在目标函数中就要用 ret 来返回，因为近调用的 call 只在栈中留下了 2 字节的返回地址（IP 寄存器的值），ret 只是从栈顶取得 2 个字节作为偏移地址载入 IP 寄存器。换句话说，如果此处偏要用 retf，CPU 也不会报错，因为 CPU 不知道之前压入栈的是什么内容，从而无法保证从栈中弹出的值其用途应该是什么。retf 会从栈中弹出 4 个字节，各 2 个字节用来替换 CS 和 IP，破坏了原 CS 寄存器的值，也破坏了栈，随后便会出错。

如果 call 是远调用，在目标函数中就要用 retf（ret far）来返回，因为远调用的 call 指令在栈中留下了段基址和段内偏移地址，retf 指令只会从栈中弹出 2 字节的偏移地址和 2 字节的段基址。同理，如果此时只用 ret 来返回，CPU 也不会报错，但栈少弹出了一个值，已经破坏了，其后的运行状况不可知。

3.2.7　实模式下的 call

在 8086 处理器中，有两个指令用于改变程序流程。一个是 jmp，另一个是 call。它们的区别是 jmp 属于一去不回头地去执行新的代码，适用环境是“交接”，如我们的 BIOS 将接力棒交给 MBR，之后再也用不到 BIOS 其余的代码。程序中有执行主线时，call 指令用于执行完一段分支后再回来的情况，当然它能回来还是需要用 ret 或 retf 来配合。

call，意为呼叫、调用。在汇编言中，用 call 命令实现一个函数的调用。

在 8086 处理器中，也就是我们所说的实模式下，call 指令调用函数有四种方式。

下面所说的前两种是近调用，后两种是远调用。

下面开始介绍第一种函数调用方式。

- 16 位实模式相对近调用

这种调用方式中，强调了两个概念，一个是近，另一个是相对。

何谓近？call 指令所调用目标函数和当前代码段是同一个段，即在同一个 64KB 的空间内，所以只给出段内偏移地址就好了，不用给出段基址。

强调一下，“近”就是指在同一段内，不用切换段，不用换段基址，只需给出段内偏移地址。

名字上和“近”有关的调用就可以用关键字 near 来修饰，near 表示在内存或寄存器中取 2 字节，这是一种数据类型转换，和数据类型伪指令 word 作用相同。near 可以省略，nasm 编译器默认在地址处取 2 字节。

何谓相对？由于是在同一个代码段中，所以只要给出目标函数的相对地址即可。

指令格式是 call near 立即数地址，注意啦，此形式中的操作数是立即数。其中的 near 可以省略。此指令是个 3 字节指令，0xe8 是此操作的操作码，占 1 字节，剩下 2 字节便是操作数。

指令中的立即数地址可以是被调用的函数名、标号、立即数，函数名同标号一样，它只是地址的人性化表示方法，最终会被编译器转换为一个实际数字地址，如 call near prog_name。不过千万不要误会编译后的操作数是目标函数 proc_name 的绝对地址，在编译后的机器码的操作数中，它是 call 指令相对于目标地址的偏移量，是个地址差。也就是说，假如 proc_name 被编译器分配的地址是 0x1234，call 指令最终的

操作数并不是 0x1234，而是目标地址减去当前 call 指令的地址，所得的差再减去此指令的长度 3，最终的结果才是 call 相对近调用指令的操作数，下面给您分析一下。

call 指令是要占用内存空间的，这里我们解释的是相对近调用，此指令机器码是 e8llhh，占用 3 字节。其中 e8 是操作码，表示相对近调用，ll 表示操作数的低位，hh 表示操作数的高位，hhll 表示跳转目标为 4 位地址。由于 x86 平台是小端字节序，故写成了 llhh，即高位在高地址，低位在低地址。这 4 位地址是个相对增量，请问这是如何得出的呢？首先用目标函数的地址减去当前 call 指令的地址，所得的差再减去此 call 指令机器码的大小（此类相对近调用的机器码大小是 3 字节，故要减去 3)，最终的结果便是 call 指令中的操作数，即与目标地址的相对地址增量。

由于此操作数并不是目标函数的绝对地址，只是相对于目标函数地址的相对增量，所以此操作数并不能直接被 CPU 使用（“直接”就是操作数以立即数的形式给 CPU 后，CPU 拿来就用，不用转换)。CPU 在实际执行中还要将此增量还原成绝对地址。所以此相对近调用并不能称为“直接”相对近调用，在下面的实例中会给大家演示这一点。

既然是相对量，就有正负之分。如果目标地址比当前 call 指令地址大，地址相对量则为正数。如果目标地址比当前 call 指令地址小，地址相对量便为负数。由此可见，操作数是个有符号数。由于段是个 16 位大小的空间，所以，正负数的范围是-32768～32767。

当在程序中调用某函数时，如 call proc_name，proc_name 是某函数名，假如 call 指令所在的地址是 0x9，而我们事先又知道 proc_name 所在的地址是 0x12，所以故做聪明地把函数调用写成 call 0x12 的形式，以为 0x12 便是 call 的操作数了，其实这样理解是错误的。那 0x12 有用没？当然有用，它被编译器拿来算偏移量了。此处的 call 是相对近转移，机器码占 3 字节，最终 call 的操作数是 0x12-0x9-3=0x6，由于是小端字节序的原因，低位在低地址，高位在高地址，进而最终的操作码是 e80600。为什么编译器要将函数的地址转换成相对地址呢？这是和硬件相关的内容了，在同一段内的函数调用（近调用)，必须要用相对地址的形式，这是硬件设计的问题，工程师们只设计了这一种形式，要想用“相对近调用”，就要迁就硬件，偏移地址或绝对地址只是个数字，从数值上无法区分这是哪类地址，硬件一律认为给它的操作数就是相对地址（即使输入的是绝对地址)。要想让 CPU 工作正确，就要确保给它输入的是真正的相对地址。如果让开发人员自己算相对地址偏移量，这太不人性化了，真要这样的话，我想这门开发语言肯定无法流行，所以编译器在编译阶段悄悄帮着开发人员把函数地址转换成适合此指令的形式，这是一个好的编译器的基本素养。

说了半天，可能您还是对我所说的持怀疑不确信的态度。咱们还是拿数据说话靠谱，让我们来证明以上是对的。

拿下面这个源码做例子。

1call.S

```
1 call near near_proc
2 jmp $
3 addr dd 4
4 near_proc：
5  mov ax， 0x1234
6   ret
```

这个函数很简单，完全是演示之用，无实际意义，就是在第一行调用了个函数，函数名叫 near_proc。此处演示的是近调用，所以“显示地”加了个 near，其实不加也是被默认处理为近调用的。

编译，nasm -o call.bin call.S 生成的二进制文件是 call.bin。

让我们直接查看 call.bin 文件中有什么，用 xxd 命令就可以啦。xxd 命令是用来逐字节查看文件的，无论什么文件在它面前都无所遁形，简直就是 Linux 界的照妖镜。由于这个命令咱们今后会常用，所以为了更方便使用，我将它封装成了个脚本，给大家贴出来看看。

```
[work@localhost book]$ cat ~/tool/xxd.sh
#usage: sh xxd.sh 文件起始地址长度
xxd -u -a -g 1 -s $2 -l $3 $1

#-u  use upper case hex letters. Default is lower case.
```

```
#
#-a | -autoskIP
#        toggle autoskIP: A single '*' replaces nul-lines.  Default off.
#
#-g bytes | -groupsize bytes
#    separate the output of every <bytes> bytes (two hex characters or eight bit-digits each)
by a whitespace.  Specify -g 0 to
#    suppress grouping.  <Bytes> defaults to 2 in normal mode and 1 in bits mode.  Grouping does
not  apply  to  postscrIPt  or
#    include style.
#
#-c cols | -cols cols
#                format <cols> octets per line. Default 16 (-i: 12, -ps: 30, -b: 6). Max 256.
#
#-s [+][-]seek
#    start at <seek> bytes abs. (or rel.) infile offset.  + indicates that the seek is relative
to the current stdin file position
#    (meaningless when not reading from stdin).  - indicates that the seek should be that many
characters from the end of
#    the input (or if combined with +: before the current stdin file position).
#    Without -s option, xxd starts at  the  current file position.
```

具体参数我就不翻译了，直接用这个脚本就行，大家把参数自己琢磨下。

脚本的使用方法是："sh 脚本被查看的文件起始地址长度"。

通过 ll -b call.bin，得到文件大小是 13 字节，于是执行：

sh～/tool/xxd.sh call.bin 0 13 回车，以下这一行是 xxd 命令的输出。

0000000: E8 06 00 EB FE 04 00 00 00 B8 34 12 C3 ....4..

以上输出中，冒号左边的 0000000 是地址，这个不重要，看冒号右边的内容。这些内容全是机器码，e80600 代表相对近调用，操作数是 0x06。各指令机器码都有自己的格式，从而确定了指令长度。例如，当 CPU 遇到 0xe8 时，就知道这是相对近调用指令，其操作数是个 2 字节的数字，总共长度是 3 字节。ebfe 是 jmp $的机器码，eb 是操作码，0xfe 是操作数，由于操作数是有符号数，所以其表示-2，而不是 254。04000000 是定义的 4 字节数据 addr dd 4。b83412 这 3 个字节中 b8 是 mov 的操作码，3412 是立即数操作数，对应的是 mov ax，0x1234。值得注意的是同样的指令，由于寻址方式不同，编译产生的机器码也是不同的。不要以为 mov 的机器码一律是 b8，这里是因为其操作数是立即数的原因，若是其他寻址方式操作码就会变的，这就是 CPU 可以通过操作码来识别操作数类型和数量的缘由。寻址方式也跟大家说过啦，这里我就不给大家举更多 mov 的例子，是时候大家自行写几个实践了，mov 指令通过组合不同的寻址方式，会产生不同的操作码。

其实这样看还不是很清楚，因为大家有可能不知道各操作码需要的操作数大小，从而无法判断指令的宽度。这好办，咱们上 bochs 看，让其边执行边反汇编给咱们看结果。下面粗体的文件是我加的注释说明。

```
<bochs:1> b 0x900 我把 call.bin 放在了内存 0x900 处,所以在此处设置断点
<bochs:2> c
(0) Breakpoint 1, 0x00000900 in ?? ()
Next at t=17827831
(0) [0x000000000900] 0000:0900 (unk. ctxt): call .+6 (0x00000909)    ; e80600
<bochs:3> trace on 用 trace on 打开了反汇编,每执行一句都会反汇编
Tracing enabled for CPU0
<bochs:4> n
(0).[17827831] [0x000000000900] 0000:0900 (unk. ctxt): call .+6 (0x00000909)    ;
e80600
上面的 call 指令,当前地址是 0x900。其操作数是正 6,这是相对地址,括号中的是目标地址的真实地址:0x909。
左边的 e80600 是 call 相对近调用的机器码,3 字节大小
这说明 call 指令的操作数确实是相对地址,即 0x909-0x900-3=0x6

(0).[17827832] [0x000000000909] 0000:0909 (unk. ctxt): mov ax, 0x1234        ; b83412
左边 b83412 是上面 mov 的机器码

(0).[17827833] [0x00000000090c] 0000:090c (unk. ctxt): ret                   ; c3
左边 c3 是 ret 的机器码

Next at t=17827834
(0) [0x000000000903] 0000:0903 (unk. ctxt): jmp .-2 (0x00000903)     ; ebfe
<bochs:5>
```

```
jmp 的操作码是 0xeb,操作数是 0xfe,即-2。
此处的 jmp 是短转移,当前指令地址 0x903 加上自己机器码的大小 2 字节,再加上操作数-2,又回到了 jmp 的地址 0x903.,
循环往复。
关于 jmp 指令,在后面无条件跳转 jmp 中有介绍。
```

这下大家相信了吧，确实是用的相对地址，程序中用的绝对地址都是给程序员看的，编译器已经将地址转换成相对地址啦。

不知道大家仔细想过这句指令没有？

0000:0900 (unk. ctxt): call .+6 (0x00000909)　　　;e80600

0000:0900 表示段寄存器 CS 为 0，段内偏移地址是 0x0900。call .+6 是相对近调用，+6 是操作数。带括号的（0x00000909），0x909 是 call 指令的目标函数的绝对地址。e80600 是机器码。

call 相对近调用中的操作数 0006 并不是被 CPU 直接用了，CPU 又将其恢复成绝对地址啦，恢复的方法是：当前的 IP 指针+操作数+机器码大小=目标函数绝对地址，也就是 0x900+0x06+0x3=0x909。所以您看，虽然操作数是地址增量，但最终，CPU 还是要想办法将其转换为绝对地址来寻址。这就是我在前面说过的，不能直接给 CPU 用，所以此调用方式不能称为“直接”相对近调用。

call 相对近调用发生时，CPU 将当前 IP 寄存器值压入栈，再把上面计算出的绝对地址载入 IP 寄存器，CPU 的航线立马被改变到目标地址处。

另外，这个输出的信息确实太多了，影响了大家的阅读。但我还是想让大家看看虚拟机 bochs 输出的原貌，这样便于大家认识调试信息，在以后的例子里为避免排版太乱，我就只拣重点信息啦。

下面是第二种函数调用方式。

- 16 位实模式间接绝对近调用

我就喜欢在名字上分析关键字，这里的关键字是 16、位、间接、绝对、近、调用。哈哈，原谅我，就像老师期末给大家画重点一样，忙活完之后，发现整本书都是重点。咱们只抓更重点的，和上一个“16 位相对近调用”相比，这种方式的区别是间接和绝对。先看看是如何体现这两词的。

“间接”是指目标函数的地址并没有直接给出，地址要么在寄存器中，要么在内存中，总之不以立即数的形式出现。

“绝对”是指目标函数的地址是绝对地址，不像“16 位相对近调用”中的那样是相对地址。

还有一点，这也是近调用，即只能调用同一个代码段中的函数，依然是只给出段内偏移就好啦，不用给出段基址。

指令的一般形式是“call 寄存器寻址”或“call 内存寻址”，如 call ax，call [0x1234]。不同的指令形式对应不同的操作码，“call 内存寻址”对应的操作码是 ff16，机器码是 ff16+16 位内存地址。机器码除了与寻址方式有关外，还和寄存器名称有关，如“call ax”的机器码是 ffd0，“call cx”的机器码是 ffd1，其他形式的机器码或操作码不单独列出。

此调用形式也是近调用，调用名称中和“近”有关的就可以用 near。near 也可以省略。由于是近调用，并没有跨段，所以 call 指令只要保留 IP 寄存器的值就好了，将其压入栈后，再用新的偏移地址替换 IP 的值。

我对任何事物都是先怀疑，经过验证之后才放心接受，为了验证上面说的可行，还是拿例子说话。上测试用例啦，以下代码纯粹演示，无实际用途，大家不要浪费时间琢磨代码背后的意思。

2call.S

```
 1 section call_test vstart=0x900
 2 mov word [addr], near_proc
 3 call [addr]
 4 mov ax, near_proc
 5 call ax
 6 jmp $
 7 addr dd 4
 8 near_proc:
 9     mov ax, 0x1234
10     ret
```

这里就演示 call 的间接近调用用法，所以大家关注第 3 行和第 5 行就好啦。第 3 行是通过内存来调用，

第 5 行是通过寄存器来调用。好吧，我知道你看到第 1 行就闷闷不乐了，那我捎带着说下。因为本测试用例要求放在内存的 0x900 处，所以加了个“vstart=0x900”。vstart 是要定义在 section 后面的，所以用了 section 关键字定义了一个节，为了表示节名不是什么神奇的东西，我随便起了个名字叫 call_test。至于为什么放在此位置，以后讲 loader 时您就清楚了。

第 7 行的 addr 是个 4 字节的变量，用来存储函数 near_proc 的地址，以供第 3 行的 call 来调用。大家发现没有，第 2 行有个关键字“word”，这个是用来告诉 CPU 一次要读写多少字节的。就拿第 2 行来说，“mov word [addr], near_proc”，near_proc 是个函数名，本身是个地址，在编译阶段就会被替换为数字，这个数字的宽度是不定的，比如 0x18 是 0x0018，还是 0x00000018 呢？这涉及到读写多少个字节的问题，这也是高级语言中数据类型的作用。由于此时是在 16 位的实模式下，我们要用 16 位的地址，必须得告诉 CPU 在此内存地址处连续读 2 字节就够了，所以用关键字 word 来表示 2 字节，即 16 位。

上面已经演示用 xxd 命令来查看二进制文件内容了，大家有兴趣还是自己验证，咱们还是实际运行看看，这样才有说服力。下面是实际执行的代码，这是从 bochs 中整理出来的，由于上一个例子中的输出信息太多了，怕大家越看越糊涂，所以这个例子中去掉了无关的信息。

以下指令中，从上到下是 2call.S 依次执行的顺序

```
mov word ptr ds：0x911， 0x0915 ；
c70611091509
//话说上边机器码好长，寻址方式变了，mov 的操作码变成了 c706。

call word ptr ds：0x911
ff161109
//用内存寻址，call 操作码是 ff16，操作数是 0x0911。call 指令在内存//0x0911 处取得目标函数地址。

mov ax， 0x1234             //函数内的指令，无关紧要
b83412                      //上面指令的机器码

ret                         //从函数中返回
c3//reg 机器码是 c3

mov ax， 0x0915             //0x0915 是函数 near_proc 的地址
                            // 现在装入 ax 寄存器
b81509                 //这是上面指令的机器码，很清楚是吧，
                       //b8 是 mov 的操作码，1509 是操作数，这是绝对地址

call ax                // ax 中是目标函数的地址，通过寄存器来寻址
ffd0                   // 操作码是 ff，操作数是 d0
                       // d0 是指 ax 寄存器，这是指令格式中的约定

//这依然是 near_proc 的函数体，再次被执行了一次
mov ax， 0x1234              ； b83412
ret                        ； c3

jmp .-2 (0x0000090f) // 回到程序主线，死循环暂停
ebfe// jmp-2 的机器码，eb 是短转移 Jmp，0xfe 是-2
```

注意，寄存器寻址中，若在寄存器名称前添加数据类型伪指令，编译器会报警告：“warning： register size specification ignored”。警告信息字面上的意思是寄存器大小被忽略。只是提示警告，不影响编译，编译的机器码依然是正确的。

数据类型伪指令有 byte、word、dword、qword 等，它们用在操作数前，相当于做数据类型强制转换，这和 C 语言中的数据类型转换是一个道理。在汇编语言中，无论操作数是立即数、寄存器，或是内存，都可以用数据类型伪指令。

near 的意思同数据类型伪指令 word 一样，是指在内存地址处取 2 字节内容，或者将操作数强制转换为 2 字节。可以认为像 near、short、far，这些用在调用或转移中的修饰符（后面会说到），其意义就是数据类型转换。每种数据类型大小不同，即表示数的范围不同，用不同的范围来表示不同的调用或转移范围。

near 若加在寄存器前面，如 call near ax，表示在 ax 寄存器取 2 字节，相当于给 ax 寄存器中的值做了类型转换。由于 near 的范围可正可负，是个有符号数范围，所以它不等同于数据类型 word。在这种情况

下，编译器发现 16 位的寄存器的值精度被破坏了（寄存器中的原值未变，被提取出来的数被强制转换了类型），被转换成其他类型，就发出个警告。如果在寄存器前加 word 就不会有警告提示，如 call word ax。

同理，后面的 far、short 也一样，far 表示取 4 字节，short 表示取 1 字节，如果在寄存器前用这些数据类型，如 call far ax 或 call short ax，编译器同样会发出这个警告。根本原因是在寄存器前添加数据类型伪指令对寄存器宽度做了强制转换，当然会发出这个警告了。大家愿意的话，就用省略 near 的形式吧。

下面介绍第三种调用方式。

- 16 位实模式直接绝对远调用

何谓直接？直接就是操作数在指令中直接给出，是立即数。

在各种转移指令中，凡是包含“直接”，都意指不需要经过寄存器或内存，操作数以立即数的形式给出。

凡是包含“远”，就意指要跨段啦，目标函数和当前指令不在同一个段中。

由于是远调用，所以 CS 和 IP 都要用新的，call 指令将来还是要回来的，所以要在栈中保留回来的路，即先把老的 CS 寄存器压入栈，再把老的 IP 寄存器压入栈后，用新的 CS 和 IP 寄存器替换，从此开启新的旅途。

指令的一般形式是：

call far 段基址（立即数）：段内偏移地址（立即数）

对于直接绝对远调用，far 也可以不加。操作码是 0x9a。机器码是 0x9a+2 字节的偏移地址+2 字节的段基址，即偏移地址在前，段基址在后，和指令的调用形式是相反的。

直接上菜。

3call.S

```
1 section call_test vstart=0x900
2 call 0：far_proc
3 jmp $
4 far_proc：
5     mov ax， 0x1234
6     retf
```

代码极其简单，直接看第 2 行，这个函数调用就用了直接绝对远调用的形式。

段基址是 0，段内偏移地址就是 far_proc。由于此类 call 是远调用，所以要和 retf 来配合，见第 6 行，用 retf 来返回。

给大家交待个背景，此程序是由 MBR 调用的，在执行此程序前，CS 的值是 0。而我们的 call 在此处用的段基址依然是 0，段基址没变。这能算跨段吗？其实 CPU 它不判断新的段值是否和当前段值一致，只要重新加载段寄存器，它就加载喽，在它的设计逻辑中未加入这一方面的思想，给它什么就是什么，虽然 CPU 是计算机的大脑，但它本身是无脑的，没有自己的思维。所以，不管当前段基址是什么，只要给它个段基址，它就接收，才不管是否和当前段重复呢。

还是让事实说话吧，上 CPU 看看再说，下面依然是从 bochs 中提炼的指令和机器码，由于本段代码较为简洁，所以在表 3-3 中列出。

表 3-3　16 位直接绝对远调用指令与机器码对照表

行　　号	地　　址	指　　令	指令机器码
1	0000：0900	call far 0000：0907	9a07090000
2	0000：0907	mov ax，　0x1234	b83412
3	0000：090a	retf	cb
4	0000：0905	jmp .-2 (0x00000905)	ebfe

见第 1 行 call far 0000：0907，call 的操作码是 9a，操作数是 07090000。在操作数中，0709，即 0x0907，是偏移地址，0000 是段基址，即偏移地址在前，段基址在后。这和汇编指令中给出的顺序是相反的，这一点要切记。

第 2、3 行的代码是 far_proc 函数的实现，其所在的绝对地址是 0000：0907。

操作数中的偏移地址 0x907，这是个绝对地址，在表 3-3 的地址列中记录的都是绝对地址。那大家看看在地址列中的第 2 行，这是 far_proc 函数的起始地址。0000：0907 表示的地址就是 0x907。这与 call 中的目标函数地址吻合，说明 call 指令确实是通过目标函数的绝对地址来调用的。

该说说第 4 种调用方式了。

- 16 位实模式间接绝对远调用

这和第 3 种的区别就是“直接”变“间接”了。也就是说，段基址和段内偏移地址，都不是立即数，要么在内存中，要么在寄存器中。可是，段基址和段内偏移地址都是 16 位地址，用一个寄存器肯定是盛不下了，至少得用两个。寄存器资源还是非常珍贵的，既然要用两个，干脆一个都不用算啦，所以这种间接绝对远调用的形式，不支持寄存器寻址，只支持内存寻址，即段基址和段内偏移地址在内存中。

16 位间接绝对远调用指令格式是：call far 内存寻址，如 call far [bx],call far [0x1234]，操作码是 ff1e。在该内存中的内容大小是 4 字节，此内容便是地址，前（低）2 字节是段内偏移地址，后（高）2 字节是段基址。在此调用方式中一定要加个关键字 far，否则就和第 2 种的间接绝对近调用一样了。

新的段基址和段内偏移既然是在内存中，访问内存的话，也要按照“段基址：段内偏移地址”的形式去操作。例如上面的 call far [0x1234]，由于没有段跨越前缀，则将默认的段基址寄存器 ds*16 后再与 0x1234 相加，得到的和为物理地址，再到该物理地址处去读取新的偏移地址和段基址，以该物理地址为起始的 2 个字节是段内偏移地址，以（该物理地址+2）为起始的 2 个字节是段基址。既然是段基址和段内偏移地址都要用新的，CPU 为了记得回来的路，先把老的 CS 寄存器压入栈，再把老的 IP 寄存器压入栈中保存起来，再用新的段基址替换 CS，新的段内偏移地址替换 IP。

以下面代码为例。

4call.S

```
1 section call_test vstart=0x900
2 call far [addr]
3 jmp $
4 addr dw far_proc， 0
5 far_proc：
6     mov ax， 0x1234
7     retf
```

第 2 行执行间接绝对远调用，addr 是个变量，在第 4 行定义的，其值的低 2 字节是函数 far_proc 的地址，高 2 位是 0，即段基址。和 call 远调用匹配的是 retf，所以第 7 行便是 retf。

好啦，直接上 CPU 运行，表 3-4 是实际执行的指令。

表 3-4　call 间接绝对远调用与 retf

行　号	地　址	指　令	指令机器码
1	0000：0900	call far ds：0x906	ff1e0609
2	0000：090a	mov ax, 0x1234	b83412
3	0000：090d	retf	cb
4	0000：0904	jmp .-2 (0x00000904)	ebfe

见表 3-4 第一行的机器码，间接绝对远调用 call 的操作码是 ff1e，retf 的操作码是 cb。

前面说过了，如果 call 不加 far 的话，就同第 2 种间接绝对近调用一样了。下面是间接绝对近调用的用法，我们同样上 CPU，对比着看一下 call 和 ret 的操作码。

```
1 section call_test vstart=0x900
2 call  [addr]
3 jmp $
4 addr dw far_proc， 0
5 far_proc：
6     mov ax， 0x1234
7     ret
```

表 3-5　　call 间接绝对近调用与 ret

行　　号	地　　址	指　　令	指令机器码
1	0000：0900	call word ptr ds：0x906	ff160609
2	0000：090a	mov ax, 0x1234	b83412
3	0000：090d	ret	c3
4	0000：0904	jmp .-2 (0x00000904)	ebfe

这次表 3-5 中 call 指令的操作码又变成了 ff16，这个操作码前面已经讲过啦，就是 16 位间接绝对近调用的操作码。ret 指令变成了 c3。

实模式下的 call 指令用法就这些了，其实咱们不必要每一种都掌握，咱们实际应用中基本上不用跨段，只用前两种近调用的形式。

3.2.8　实模式下的 jmp

无条件跳转，是指“生硬地”改变 CPU 航线，将程序流转移到新的位置。前面说过啦，CPU 的航线是段寄存器 CS 和 IP，所以 jmp 指令也是通过修改这两个寄存器来为 CPU 导航的。

jmp 转移指令只要更新 CS：IP 寄存器或只更新 IP 寄存器就好了，不需要保存它们的值，所以跳到新的地址后没办法再回来，它属于“一去不回头”地去执行新指令。

和 call 一样，按远近（是否跨段）来划分，大致分为两类，近转移、远转移。不过在转移方式中，还有个更近的，叫短转移。

一共有 5 类转移方式，下面开始介绍第一种。

- 16 位实模式相对短转移

其实在介绍 call 指令的时候我们就已经见识过相对短转移了，就是那个常用的 jmp $。指令格式是 jmp short 立即数地址。

此处的立即数地址也可以是标号，因为标号只是更为人性化的立即数形式，在编译阶段将被分配为某个地址。

和 call 指令一样，既然此转移方式是“相对”，也就意味着操作数是个相对增量，所以其有正负之分。也就是说操作数是个有符号数。相对短转移的机器码大小是 2 字节，操作码是 0xeb，可知其为 1 字节大小。那操作数占多少字节？答案显然啦，是 2-1=1 字节。这正是我想说的，相对短转移中的“短”，体现在操作数中，即跳转的范围只能是 1 字节有符号数所表示的范围，即-128～127。

短转移，意味着只在段内转移，不需要跨段，所以只需要偏移地址就够了，当然，此偏移地址并不是直接给出的，而是经操作数转换而成的。话说欲知如何转换，还得先知道操作数是怎么来的，因为转换过程是获得操作数的逆运算。

也不用卖关子了，还记得之前讲 call 的相对近调用形式时所说的操作数吗?写在代码中的操作数地址并不是真正机器码中的操作数，是经过编译器处理成与目标地址的地址差再减去机器码大小。在 jmp 的相对短转移形式中也是一样的，操作数也要经过编译器转换，其转换方法和 call 的相对近调用是一样的原理，即用跳转后的目标地址减去当前地址，所得的差再减去此种 jmp 指令的大小 2 字节，最终的结果便是此 jmp 相对短转移的操作数。

值得注意的是和 call 相对近调用一样，此立即数地址（操作数）CPU 并不能直接用，它只是个地址差（目标地址减去 jmp 所在地址，再减去 jmp 指令机器码大小 2 字节），所以此相对短转移方式中，并没有“直接”二字。而 CPU 是要用绝对地址来寻址的，方法是将此 jmp 的操作数加上寄存器 IP 的值，再加上 2 字节，所得的结果便是目标地址的绝对地址，这样的地址 CPU 才能用。CPU 把求得的绝对地址载入 IP 寄存器，由于这是短转移，目标地址和当前指令在同一个段内，所以 CS 段寄存器不用修改，CPU 就实现了向新位置的转移。

另外，关键字 short 是指明让 nasm 编译器将 jmp 编译为相对短转移的形式，如果条件不满足 short 的要求，即操作数大小不满足-128～127 的范围，则会编译失败。此参数可以省略，但省略后并不能保证 nasm 依然把它编译成相对短转移的形式，也许能，也许不能。对于 nasm 来说，把它编译为何种形式的转移指

令，取决于操作数的大小，在这先埋下伏笔，等介绍下一种“相对近转移”时大家就明白了。

咱们边说边看实例，上个小例子，如下面代码。

1jmp.S

```
1section call_test vstart=0x900
2 jmp short start
3 times 127 db 0
4 start：
5     mov ax， 0x1234
6     jmp $
```

第 2 行的 jmp short start 采用短转移方式。目标地址是第 4 行的 start。

在第 3 行特意定义了 127 字节的数据，目的是用来间隔目标地址 start，使第 2 行的 jmp 和第 4 行的 start 保持一定距离。这 127 字节实际上就是 jmp 短转移方式的操作数，即第 4 行 start 的地址减去第 2 行 jmp short start 的地址后，再减去 2 字节等于 127。

简短介绍之后，还是上 CPU 测试一下，老样子，依然是从虚拟机中提炼重点信息，见表 3-6。

表 3-6　　jmp 相对短转移指令

行　号	地　址	指　令	机　器　码
1	0000:0900	jmp .+127 (0x00000981)	eb7f
2	0000:0981	mov ax, 0x1234	b83412
3	0000:0984	jmp .-2 (0x00000984)	ebfe

第 1 行，在地址 0x900 处的指令是 jmp .+127，操作数是 127，其转移的真实地址是 0x981，前面说过啦，操作数是不能直接给 CPU 用的，CPU 要将其转换成绝对地址。所以此地址是这样得来的：0x900+2+127=0x981。

机器码 eb7f。由于此指令是 2 字节大小，所以第 1 个字节 eb 是操作码，第 2 个字节 7f 是操作数，即十进制的 127。

第 2 行的就不用解释啦，前面在演示 call 指令时有说过。

第 3 行的 jmp .-2 其实就是编译前的 jmp $，同第 1 行一样是个相对短跳转，原理是一样的，其最终也要将操作数-2 转换为绝对地址 0x984。转换过程是当前 IP 寄存器的值（jmp 地址 0x984）+ 操作数负 2 + 此指令大小 2 字节=0x984。

前面说过啦，操作数范围是-128～127，如果操作数不在此范围，将会在编译阶段报错。给大家实际演示一下出错情况，比如下面这段代码。

1.1jmp.S

```
1 section call_test vstart=0x900
2 jmp short start
3 times 128 db 0
4 start：
5     mov ax, 0x1234
6     jmp $
```

由于在第 3 行我们定义了 128 字节的数据，这样第 2 行的 jmp 所在地址和第 4 行的 start 所在地址，其地址差便为 130 字节（如前所述，jmp short start 本身占 2 字节，之间还有 128 个 1 字节，共 130 字节）。而操作数便是 128 字节，操作数超过了范围，编译时将会报以下错误。

nasm -o 1jmp.bin 1jmp.S 回车后，nasm 将会报错如下。

```
1jmp.S:2: error: short jump is out of range
```

除了将操作数改为-128～127 之间外，难道就没有别的办法吗？只能转移到这么近吗？必须有更好的方法。

解决的办法无外乎就是将操作数的范围增大，突破 1 字节的有符号数表示范围就行了。

（1）将第 2 行 jmp 后的 short 去掉，改成 near。

（2）第 2 行 jmp 后什么都不写，让 nasm 编译器来自动判断，用 short，还是 near。

此处提到的 near 又是什么？欢迎 jmp 的第二种用法，相对近转移。

- 16 位实模式相对近转移

前面给大家埋下了近转移的伏笔，在此揭晓谜底啦。

说重点，相对近转移和相对短转移相比，就是操作数范围增大了，由 8 位宽度变成了 16 位宽度，操作数依然是地址相对量，可正可负，范围是-32768～32767。由此可见，概念上“近”比“短”表示的范围更远一些。

指令格式是 jmp near 立即数地址，其操作码是 0xe9。

指令中的立即数地址也要经过编译器转换为地址偏移量，再变成机器指令中的操作数。

无论是 call，还是 jmp，介绍了这么多“相对”的形式，大家早就看出来了，这个操作数都是这样来的，即目标地址减去当前指令地址后所得的差，再减去机器码大小。

由于相对近转移的机器码是 3 字节，所以操作数=目标地址-当前指令地址-3。

还是那句话，由于此转型中包括关键字“相对”，机器码中的操作数是个地址增量，所以 CPU 要将其还原成绝对地址。因为“近”转移不需要跨段，所以只算出偏移地址就对啦，于是，绝对地址=操作数+IP 寄存器的值+3。

之后，CPU 用此绝对地址替换 IP 寄存器中的值，由于这是近转移，目标地址和当前指令在同一个段内，所以 CS 段寄存器不用修改，CPU 马上就飞到新的目标地址了。

有了这种近转移的方法，咱们看看能否解决上一个短转移因为操作数超过范围的报错。

修改上面出错的源文件 1.1jmp.S 中的第 2 行代码，将 short 改为 near，如 2jmp.S。

2jmp.S

```
1 section call_test vstart=0x900
2 jmp near start
3 times 128 db 0
4 start：
5     mov ax, 0x1234
6     jmp $
```

只是第 2 行改为了 jmp near start，其他代码都不变。

nasm -o 2jmp.bin 2jmp.S 回车，没有任何输出，这表示编译成功了。表 3-7 是源程序 2jmp.S 上 CPU 后的情况。

表 3-7　　jmp 相对近转移指令

行　　号	地　　址	指　　令	机　器　码
1	0000:0900	jmp .+128 (0x00000983)	e98000
2	0000:0983	mov ax, 0x1234	b83412
3	0000:0986	jmp .-2 (0x00000986)	ebfe

大家看表 3-7 的第 1 行，在地址 0x900 处的指令是 jmp.+128，括号中的 0x983 是其要跳转的绝对地址，也就是会跳到第二行的 mov ax，　0x1234。0x983 是怎么来的呢？前面说过啦，操作数（128）+IP 寄存器值（0x900）+机器码大小（3）。E98000 是 jmp.+128 的机器码，e9 是操作码，8000 是操作数，由于 x86 是小端字节序形式，所以其值为 0x0080，即十进制的 128。第三行的 jmp.-2，也就是源文件中的 jmp$，被编译为短转移形式，其操作码是 0xeb。

补充一下，按照目前 2.10.07 版本的 nasm，如果超过了 16 位有符号数的范围-32768～32767，编译器并不会报错，只是会将超过 16 位的部分忽略，只保留 16 位的结果。

jmp“相对”转移的形式介绍完了，分别是相对短转移和相对近转移。其中的 short 和 near 如果省略，nasm 编译器会根据目的地址和当前地址的偏移量大小来自行判断，若偏移量属于范围-128～127，则编译为 short 短转移形式。若超过了短转移的范围就编译为 near 近转移形式。

下面介绍第三种形式，它是咱们要介绍的最后一种近转移，操作数不再是“相对”地址偏移量，而是

绝对地址啦。欢迎“间接绝对近转移”。

- 16 位实模式间接绝对近转移

同上一个“jmp 相对近转移”相比，本“间接绝对近转移”其目标地址是绝对地址，并且未在指令中直接给出，存在寄存器或内存中。

在讲述 call 指令的调用方式时，我们说过了“间接”的意思。间接，是指操作数并不直接给出，而是存储在寄存器或内存中。绝对地址顾名思义，就是段内偏移地址，指的是“CS：IP”中的 IP 值，偏移地址是 16 位。经过这样的拆词分析，概念已经没法再清楚了。所以，“间接绝对近转移”就是指段内转移，转移的地址是 16 位宽度，也就是 2 字节，地址要么在寄存器中，要么在内存中。

和“近”有关的转移就可以用关键字 near 来修饰，表示在内存或寄存器中取 2 字节，这是一种数据类型转换，在此，near 依然可以省略，nasm 编译器默认在地址处取 2 字节。

指令格式是 jmp near 寄存器寻址，或者 jmp near 内存寻址。

若操作数在内存中，在不使用段跨越前缀的情况下，段基址寄存器是 DS。

由于这也是近转移，CS 寄存器的值不用修改，CPU 只要用 16 位寄存器的值或内存中的 2 字节载入 IP 寄存器，CPU 马上就被带到新的地址。

采用寄存器寻址的 jmp 指令，其操作码是 0xff，操作数随寄存器的不同而不同。采用内存寻址的 jmp 指令，其操作码还要看段基址寄存器用的是哪个，见下面实例吧。

在通过内存寻址时，在不用段跨越前缀的情况下，默认需要用到 DS 寄存器。

老规矩，上代码验证一下。

3jmp.S

```
1 section call_test vstart=0x900
2 mov ax， start
3 jmp near ax
4 times 128 db 0
5 start：
6     mov ax， 0x1234
7     jmp $
```

在 3jmp.S 第 2 行将地址 mov 到寄存器 ax 中，再通过 jmp near ax 实现转移。

说到这里，在上一节中所说的 call 调用形式中，第二种调用形式间接绝对近调用，里面出现了 near 小插曲，near 放在寄存器前编译器就会报警告：“warning： register size specification ignored”，这里也出现了。如果您仔细看过那段小插曲的推测部分，就会了解，near 可能（我用的是可能）只是个数据类型，和 byte、word、dword 作用一样，相当于强制数据类型转换，用来控制数值范围，从而控制了转移的范围。

表 3-8　　jmp 间接绝对近转移之寄存器

行　号	地　址	指　令	机　器　码
1	0000:0900	mov ax, 0x0985	b88509
2	0000:0903	jmp ax	ffe0
3	0000:0985	mov ax, 0x1234	b83412
4	0000:0988	jmp .-2 (0x00000988)	ebfe

表 3-8 中第 2 行的 jmp ax 便是源代码中的 jmp near ax，机器码是 ffe0，操作码是 0xff，e0 代表寄存器 ax。其他不用多解释了，大家看得太多了。

再举个 jmp 用内存寻址的例子。

3.1jmp.S

```
1 section call_test vstart=0x900
2 mov word [addr]， start
3 jmp near [addr]
4 times 128 db 0
5 addr dw 0
6 start：
```

```
7     mov ax， 0x1234
8     jmp $
```

代码 3.1jmp.S 中的第 2 行，将 start 地址 mov 到内存[addr]，addr 是个变量，2 个字节大小，在第 5 行定义。将目标地址写入变量 addr 后，在第 3 行，jmp 通过此变量来转移。

表 3-9 是其实际执行时的情况。

表 3-9　　jmp 间接绝对近转移之内存

行　号	地　址	指　令	机　器　码
1	0000:0900	mov word ptr ds:0x98a, 0x098c	c7068a098c09
2	0000:0906	jmp word ptr ds:0x98a	ff268a09
3	0000:098c	mov ax, 0x1234	b83412
4	0000:098f	jmp .-2 (0x0000098f)	ebfe

表 3-9 的第 1 行是将立即数写入内存，对应源码 3.1jmp.S 的第 2 行。大家看下它的机器码有多复杂。操作数是 c706，这还是默认数据段 ds 寄存器，如果使用段跨越前缀，操作码又会变成其他样子了。表 3-9 的第 2 行是 jmp 通过内存寻址实现转移，对应源码第 3 行，near 在此被替换为 word。

近一步证明 near 和 short 就是 nasm 数据类型伪指令，不同的数据类型有不同的宽度，通过转换数据宽度来实现转移范围。

- 16 位实模式直接绝对远转移

经过前面拆词分析的不断强化，大家一看这个名字，基本就知道我想说什么了。直接绝对远转移中：

"直接"是指操作数不仅是立即数，而且 CPU 直接拿来就用，不用再转换。

"绝对"是指提供的操作数是绝对地址。

"远"是指目的地址和当前指令不是一个段，有跨段的需求，所以要操作数要包括新的段基址和段内偏移。

直接绝对远转移就是以立即数的形式给出目标地址的段基址和段内偏移地址。指令格式为 jmp 立即数形式的段基址：立即数形式的段内偏移地址。例如 jmp 0： 0x900，其中 0 是段基址，0x900 是段内偏移地址。

由于是远转移，所以 CPU 用操作数中的段基址载入 CS 寄存器，用操作数中的偏移地址载入 IP 寄存器后才完成转移。

还是拿例子说话，代码如下。

4jmp.S

```
1 section call_test vstart=0x900
2 jmp 0：start
3 times 128 db 0
4 start：
5     mov ax， 0x1234
6     jmp $
```

4jmp.S 的第 2 行 jmp 0：start，段基址为 0，偏移地址是函数符号 start，最终也会被编译器编译为数字地址。其他的也没啥可说的啦，直接看实际运行情况。

表 3-10　　jmp 直接绝对远转移

行　号	地　址	指　令	机　器　码
1	0000:0900	jmp far 0000:0985	ea85090000
2	0000:0985	mov ax, 0x1234	b83412
3	0000:0988	jmp .-2 (0x00000988)	ebfe

大家看表 3-10 第一行，在内存地址 0x900 处的指令是 jmp far 0000：0985，这对应源码 4jmp.S 的第 2 行。机器码是 ea85090000，操作码是 0xea。注意，在指令中的操作数顺序与机器码的操作数顺序是相反的，指令中的操作数顺序是编译器为人使用方便而规范的，人在感观上觉得段在前，偏移地址在后，比较符合自然习惯。而机器码中的偏移地址在前，段基址在后，目的一是为指令格式较统一，指令中固

定字段是固定内容。二是硬件电路为了高效而故意设计成这样。毕竟大多数情况下都是段内转移，在译码阶段，得到操作码后，紧接着的是段内偏移地址，不是更高效吗？

- 16 位实模式间接绝对远转移

和上一个“直接绝对远转移”比，此处操作数是由“直接”变“间接”，不再拆词分析啦，也就是说，操作数不是直接给出的，即段基址和段内偏移不是立即数的形式。由于操作数是两个数，放在一个寄存器中是不可能的啦，前面在说 call 间接绝对远调用时也分析过啦，操作数只能放在内存中。为了指示 CPU 在内存中取 4 个字节，需要在指令中用关键字 far，即前两个字节是段内偏移地址，后两个字节是段基址。若不指定，则和第三种的“间接绝对近转移”一样，只在内存处取 2 字节。

所以其指令格式是：jmp far 内存寻址。

由于操作数在内存中，在不使用段跨越前缀的情况下，段基址寄存器是 DS。

此指令的操作数，需要访问内存才能得到，所以需要知道寻址方式。机器码与寻址方式有关，要根据实际使用的数据段寄存器等情况来决定。

同样，由于是远转移，CPU 的 CS 寄存器和 IP 寄存器都要修改成操作数中指定的值，从而实现转移。

上菜，见代码 5jmp.S。

5jmp.S

```
1 section call_test vstart=0x900
2 jmp far [addr]
3 times 128 db 0
4 addr dw start，0
5 start：
6     mov ax， 0x1234
7     jmp $
```

在 5jmp.S 中的第 4 行定义了地址变量 addr，此变量的低 2 字节是偏移地址，高 2 字节是段基址。第 2 行直接通过此变量 addr 访问跳转地址。

表 3-11 是实际运行的情况。

表 3-11　间接绝对远转移

行　号	地　址	指　令	机　器　码
1	0000:0900	jmp far ds:0x984	ff2e8409
2	0000:0988	mov ax, 0x1234	b83412
3	0000:098b	jmp .-2 (0x0000098b)	ebfe

表 3-11 第一行，地址 0x900 处的指令是 jmp far ds：0x984，意思是在地址 0x984 处取得跳转目标地址。由于地址 0x984 中的内容是普通数据，不是指令，所以未在表 3-11 中列出。机器码 ff2e8409，其中的 2e 代表 DS 寄存器。

好啦，无条件跳转指令介绍完了，有没有觉得好长啊？

3.2.9　标志寄存器 flags

按理说，既然有“无条件转移”，就应该有“有条件转移”，真实情况也确实是这样。讲完了无条件转移指令后，该到有条件转移指令啦，可是我们得知道这个条件在哪里，是什么条件。这样我们才能根据这些条件做出是否转移的判断。

这些条件就放在了标志寄存器 flags 中。在名字上看，flag 加了 s，说明是 flag 的复数形式，是 flag 的集合，在此寄存器中有很多标志位。

实模式下标志寄存器是 16 位的 flags，在 32 位保护模式下，扩展（extend）了标志寄存器，成为 32 位的 eflags。

其实这些用于判断的条件，本质上是上一条指令执行的结果。想想看，我们在实际编程中做出的条件判断，也是判断上一句代码的结果。例如 if、switch 等，它们都是判断一个已存在的数据结构的结果。

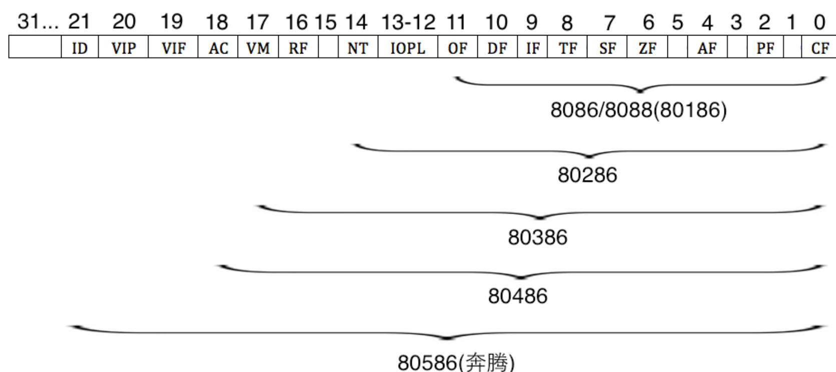
显而易见，“条件”得存在于计算机系统的某个地方，我们才能访问到它。

在 c 等高级语言中，这些结果（条件）放在了内存变量中，我们判断的地方是某个内存单元。但在机器世界里，它不像 C 语言这般高级，所谓的“高级”并不是说真的有多高大上，“高级”指的是比较接近人类语言，用起来比较方便，更靠近人的逻辑思维，不用考虑过多细节的宏观抽象。但越是“高级”，越是受限，比如想盖个大楼，高级语言给咱们提供好了各种成型的建材，如彩钢、楼板、水泥，只能用提供好的材料去施工，由于材料早已经准备好了，所以建房速度就是快，但这些材料只能盖楼房。而低级语言如汇编，给咱们提供的是水、土、沙、金属等最基本的原料，我们可以随意加工成我们需要的建筑材料，这些材料想盖别墅盖别墅，想盖瓦房盖瓦房，形式不受限，但由于是自己去制造建筑材料，房子盖得就很慢。越底层的东西往往灵活性越强，提供的信息越丰富，它不单纯记录结果，而且还能告诉你这个结果来的过程。由于提供的信息很丰富，就用一个寄存器来集中这些信息，这就是 flags 寄存器。flags 寄存器中存储的信息，只是结果的特征，即标志，并不是真正的结果，结果可以存储在内存中。例如 64*2 的结果是 128，128 不会直接存到寄存器，真要是每个结果都用寄存器存放，那得用多少寄存器。所以 flags 寄存器是用于记录结果的特征标志，这些标志告诉大家，为了产生这个结果，机器都做了什么。比如，有时单纯的一个结果并不能让我们了解数据的全貌，产生这个结果的过程中是否有溢出，如果不知道这些，怎么确定结果是正确的呢？

IA32 指令中并没有提供高级逻辑的指令，但无论逻辑多复杂，都可以通过最简单的判断和转移来实现。判断哪里？判断什么？这个判断的对象就是标志寄存器中的标志位。

flags 寄存器是 16 位宽，保护模式下对其扩展（extend）成 32 位的 eflags 寄存器。

由于 eflags 寄存器兼容 flags 寄存器，还是给各位看官呈上 eflags，如图 3-10 所示。



▲图 3-10　eflags 寄存器

咱们不用把所有的标志位都学了，只要把基本的搞清楚就成。毕竟本书不是讲汇编语言，下面给大家来个笼统地介绍。

以下标志位仅在 8088 以上 CPU 中有效。

第 0 位的是 CF 位，即 Carry Flag，意为进位。运算中，数值的最高位有可能是进位，也有可能是借位，所以 carry 表示这两种状态。不管最高位是进位，还是借位，CF 位都会置 1，否则为 0。它可用于检测无符号数加减法是否有溢出，因为 CF 为 1 时，也就是最高位有进位或借位，肯定是溢出。

再说点没用的，第 1、3、5、15 位没有专门的标志位，空着占位用。

第 2 位为 PF 位，即 Parity Flag，意为奇偶位。用于标记结果低 8 位中 1 的个数，如果为偶数，PF 位为 1，否则为 0。注意啦，是最低的那 8 位，不管操作数是 16 位，还是 32 位。奇偶校验经常用于数据传输开始时和结束后的对比，判断传输过程中是否出现错误。

第 4 位为 AF 位，即 Auxiliary carry Flag，意为辅助进位标志，用来记录运算结果低 4 位的进、借位情况，即若低半字节有进、借位，AF 为 1，否则为 0。

第 6 位为 ZF 位，即 Zero Flag，意为零标志位。若计算结果为 0，此标志为 1，否则为 0。

第 7 位为 SF 位，即 Sign Flag，意为符号标志位。若运算结果为负，则 SF 位为 1，否则为 0。

第 8 位为 TF 位，即 Trap Flag，意为陷阱标志位。此位若为 1，用于让 CPU 进入单步运行方式，若为 0，则为连续工作的方式。平时我们用的 debug 程序，在单步调试时，原理上就是让 TF 位为 1。可见，软件上的很多功能，必须有硬件的原生支持才能得以实现。

第 9 位为 IF 位，即 Interrupt Flag，意为中断标志位。若 IF 位为 1，表示中断开启，CPU 可以响应外部可屏蔽中断。若为 0，表示中断关闭，CPU 不再响应来自 CPU 外部的可屏蔽中断，但 CPU 内部的异常还是要响应的，因为它关不住。

第 10 位为 DF 位，即 Direction Flag，意为方向标志位。此标志位用于字符串操作指令中，当 DF 为 1 时，指令中的操作数地址会自动减少一个单位，当 DF 为 0 时，指令中的操作数地址会自动增加一个单位，意即给地址的变化提供个方向。其中提到的这个单位的大小，取决于用什么指令。

第 11 位为 OF 位，即 Overflow Flag，意为溢出标志位。用来标识计算的结果是否超过了数据类型可表示的范围，若超出了范围，就像水从锅里溢出去了一样。若 OF 为 1，表示有溢出，为 0 则未发生溢出。专门用于检测有符号数运算结果是否有溢出现象。

以下标志位仅在 80286 以上 CPU 中有效。相对于 8088，它支持特权级和多任务。

第 12～13 位为 IOPL，即 Input Output Privilege Level，这用在有特权级概念的 CPU 中。有 4 个任务特权级，即特权级 0、特权级 1、特权级 2 和特权级 3。故 IOPL 要占用 2 位来表示这 4 种特权级。如果您对此感到迷茫，不用担心，这些将来咱们在保护模式下也得实践。

第 14 位为 NT，即 Nest Task，意为任务嵌套标志位。8088 支持多任务，一个任务就是一个进程。当一个任务中又嵌套调用了另一个任务（进程）时，此 NT 位为 1，否则为 0。

以下标志位仅用于 80386 以上的 CPU。

第 16 位为 RF 位，即 Resume Flag，意即恢复标志位。该标志位用于程序调试，指示是否接受调试故障，它需要与调试寄存器一起使用。当 RF 为 1 时忽略调试故障，为 0 时接受。

第 17 位为 VM 位，即 Virtual 8086 Model，意为虚拟 8086 模式。这是实模式向保护模式过渡时的产物，现在已经没有了。CPU 有了保护模式后，功能更加强大了，但为了兼容实模式下的用户程序，允许将此位置为 1，这样便可以在保护模式下运行实模式下的程序了。实模式下的程序不支持多任务，而且程序中的地址就是真实的物理地址。所以在保护模式下每运行一个实模式下的程序，就要为其虚拟一个实模式环境，故称为虚拟模式。

以下标志位仅用于 80486 以上的 CPU。

第 18 位为 AC 位，即 Alignment Check，意为对齐检查。什么是对齐呢？是指程序中的数据或指令其内存地址是否是偶数，是否是 16、32 的整数倍，没有余数，这样硬件每次对地址以自增地方式（每次自加 2、16、32 等）访问内存时，自增后的地址正好对齐数据所在的起始地址上，这就是对齐的原理。对齐并不是软件逻辑中的要求，而是硬件上的偏好，如果待访问的内存地址是 16 或 32 的整数倍，硬件上好处理，所以运行较快。若 AC 位为 1 时，则进行地址对齐检查，为 0 时不检查。

以下标志位只对 80586（奔腾）以上 CPU 有效。

第 19 位为 VIF 位，即 Virtual Interrupt Flag，意为虚拟中断标志位，虚拟模式下的中断标志。

第 20 位为 VIP 位，即 Virtual Interrupt Pending，意为虚拟中断挂起标志位。在多任务情况下，为操作系统提供的虚拟中断挂起信息，需要与 VIF 位配合。

第 21 位为 ID 位，即 Identification，意思为识别标志位。系统经常要判断 CPU 型号，若 ID 为 1，表示当前 CPU 支持 CPU id 指令，这样便能获取 CPU 的型号、厂商等信息。若 ID 为 0，则表示当前 CPU 不支持 CPU id 指令。

其余剩下的 22～31 位都没有实际用途，纯粹是占位用，为了将来扩展。

了解了这个标志位后，一些“条件”指令便有法可依啦，比如下面要说的，有条件转移。

3.2.10 有条件转移

有条件转移不是简单的一个指令，它是一个指令族，我们在此简单称 jxx。如果条件满足，jxx 将会跳转到指定的位置去执行，否则继续顺序地执行下一条指令。

其格式为 jxx 目标地址。若条件满足则跳转到目标地址，否则顺序执行下一条指令。

其中，目标地址只能是段内偏移地址。在实模式下，由编译器根据当前指令与目标地址的偏移量，自行将其编译成短转移或近转移。在保护模式下，寄存器中宽度已经到了 32 位，32 位的偏移地址可以访问到整个 32 位地线总线的 4GB 内存空间，编译器不再区分转移方式。

进行条件转移，所谓的条件就是判断上一条指令的结果是否满足某方面或某些方面，能够影响标志位的指令才能被其后的条件指令用作条件。所以条件转移指令一定得在某个能够影响标志位的指令之后进行。也就是说，每执行一条指令，标志寄存器中的相应位都会记录这条指令所带来的变化。所以说，条件转移指令，判断的就是上一条指令对标志位的“影响”，这些“影响”就是条件。

条件转移指令中所说的条件就是指标志寄存器中的标志位。jxx 中的 xx，就是各种条件的分类，每种条件有不同的转移指令。下面将条件展开，将各指令实例化列出，见表 3-12。

表 3-12　条件转移指令

转移指令	条　　件	意　　义	英文助记
jz/je	ZF=1	相减结果等于 0/相等时转移	Jump if Zero/Equal
jnz/jne	ZF=0	不等于 0/不相等时转移	Jump if Not Zero/ Not Equal
js	SF=1	负数时转移	Jump if Sign
jns	SF=0	正数时转移	Jump if Not Sign
jo	OF=1	溢出时转移	Jump if Overflow
jno	OF=0	未溢出时转移	Jump if Not Overflow
jp/jpe	PF=1	低字节中有偶数个 1 时转移	Jump if Parity/Parity Even
jnp/jpo	PF=0	低字节中有奇数个 1 时转移	Jump if Not Parity/Parity Odd
jbe/jna	CF=1 或 ZF=1	小于等于/不大于时转移	Jump if Below or Equal/Not Above
jnbe/ja	CF=ZF=0	不小于等于/大于时转移	Jump if Not Below or Equal/Above
jc/jb/jnae	CF=1	进位/小于/不大于等于时转移	Jump if Carry/Below/Not Above Equal
jnc/jnb/jae	CF=0	未进位/不小于/大于等于时转移	Jump if Not Carry/Not Below/Above Equal
jl/jnge	SF!=OF	小于/不大于等于时转移	Jump Less/Not Great Equal
jnl/jge	SF=OF	不小于/大于等于时转移	Jump if Not Less/Great Equal
jle/jng	ZF!=OF 或 ZF=1	小于等于/不大于	Jump if Less or Equal/Not Great
jnle/jg	SF=OF 且 ZF=0	不小于等于/大于时转移	Jump Not Less Equal/Great
Jcxz	CX 寄存器值=0	cx 寄存器值为 0 时转移	Jump if register CX`s value is Zero

有没有觉得好多好乱好烦？这里面同义的好多啊，比如 jl 和 jnge，直接就理解为“小于时转移”就成了，何必再弄个同义词 jnge“不大于等于时转移”呢？其实不用那么闹心，经常用的就两三个。而且，这些转移指令是由意义明确的字符拼成的。

a　表示 above
b　表示 below
c　表示 carry
e　表示 equal
g　表示 great
j　表示 jmp
l　表示 less
n　表示 not
o　表示 overflow
p　表示 parity

根据这些缩写，从字面上就大概了解具体转移指令的意义了。经验表明，即使不理解的东西，时间一长也会理所当然就这样，对于新生事物，只是时间问题。

好啦，对于条件转移指令，咱们差不多就到这了，咱们得开足马力介绍操作系统啦。在这里小弟诚恳地跟大家建议，如果汇编基础薄弱，还是专门去看看专讲汇编的书，本书真地不能做到从 0 开始讲汇编语言。

3.2.11 实模式小结

作为您的小伙伴，对于汇编语言中的部分，我也就能帮您到这儿了，毕竟本书不是讲汇编语言的。另外本书中用到的汇编其实不深奥，我尽量给大家说清楚，如果汇编基础实在薄弱，让我从头讲汇编的话，兄弟我实在是做不到啊。说句负责任的话，即使在这里给大家讲解汇编语言，也是假设大家有一定基础，假设大家曾经学过，现在只是概念模糊了，相当于是给大家"复习"汇编语言。若一点基础都没有，还是建议您暂时把本书放一放，先找本专门的汇编书看看，学得不用太深，书中用汇编的地方主要就是 MBR 和以后要讲的内核加载器 loader，用的还都是较浅显的指令，有点基础就能看懂了。

说句良心话，学习任何知识都没有真正从零基础开始的，这就是本书不敢标榜从零基础学习的原因。新知识的学习是基于脑子中旧有的知识体系，它是旧有知识的扩充与延伸，我们在学习新知识之初，肯定是要找出旧知识间的相互关联，并用这种关联去推导出新的知识。这就是学习一门新课程前所需要的基础知识，如同上大学要有高中的基础一样。只有用旧知识能够讲得通，新知识才能够被理解，被接受，被吸收，知识本身是个不断迭代的过程。如果一本计算机书能够让搞艺术的人（如玩音乐的、画画的、搞服装设计的）看懂，那也只是"可能"做到了零基础。

咱们的实模式到这就告一段落。将来咱们还是以保护模式为主，实模式有很多不靠谱的地方，终将被保护模式替代。

实模式被保护模式淘汰的原因，最主要是安全隐患。

在实模式下，用户程序和操作系统可以说是同一特权的程序，因为实模式下没有特权级，它处处和操作系统平起平坐，所以可以执行一些具有破坏性的指令。

程序可以随意修改自己的段基址，这样便在 1MB 的内存空间内不受阻拦，可以随意访问任意物理内存，包括访问操作系统所在的内存数据。这就给程序员开放了无限的自由，程序员访问内存可以说是指哪打哪。

由于完全没有保护性可言，用户程序甚至可以覆盖操作系统在内存中的映像，整个计算机世界的和平全靠程序员的心情。

3.3 让我们直接对显示器说点什么吧

在上一章中，我们给出了 mbr 的简单实现，让它飞了一会。不过由于我当时还没有和大家介绍如何直接通过显卡来输出字符，所以我们用的是 BIOS 提供的 0x10 中断来实现的滚屏和打印字符。

但由于以下原因，我们不得不向 BIOS 说拜拜，是时候打破同硬件打交道的恐惧与神秘了。

3.3.1 CPU 如何与外设通信——IO 接口

介绍显卡之前，必须得和大家交待清楚，那么多的外部设备，CPU 是如何与它们交流的。

大家都学过微机接口技术吧？没学过也没关系，反正我也只是笼统地说说，保证大家一定能看得懂。

按理说，如果硬件种类较少，让 CPU 直接同硬件进行 IO 操作也不是很过分，但现实不是这样的，计算机能发展到今天这样兴盛，是和诞生各种各样的硬件分不开的。微型计算机通过外部设备与外面的世界互换信息，外部设备种类繁多，原理各异，有机械式、电动式、电子式，输出的信号也多种多样，有模拟量、数字量、开关量。它们都有自己的特性，数据格式不相同，有的外设用串行数据，有的是并行数据，并且它们都在自己的时序下工作，无论它们的速度如何，在 CPU 看来都太慢了。

让 CPU 小朋友与每个"个性不同，脾气迥异"的硬件大大们打交道，这也太为难 CPU 了，您看，通过执行 jmp $这样的死循环语句就能看得出，人家 CPU 可是个踏实低调的主，所以，"交际"这类活动对它还是少点好。再说，同任何一个设备打交道，CPU 那么速度那么快，它不得嫌弃别人慢吗，为了减少自己的等待时间，还得为低速设备准备数据缓冲区。CPU 用的信号都是 TTL 电平，外设大多数都是机电

设备，机电设备可不能用 TTL 电平驱动，这还不算完呢，CPU 系统总线上传送的都是并行数据（所以你听到的都是 8 位、16 位、32 位 CPU……），外设可是并行、串行都有，还得转换格式，想想就麻烦啊。看来，不可能让 CPU 一一适应它们，否则 CPU 要做的工作太多了。

CPU 面临的问题，就像校长面临一群学生一样，让校长亲自管理每个学生的学习，即使是肌肉男施瓦辛格也得累倒，于是，班主任的出现帮了大忙，每个班主任负责一批学生，由他们了解学生的情况后再向校长汇报，这样校长他不需要过人的体格，工作起来也会游刃有余了。人创造出来的东西必然脱离不了人的思维，CPU 工程师们也给 CPU 找了“班主任”，在 CPU 和外设之间加了个代理，总之，以后 CPU 有什么事就同它接触就行了。什么速度不匹配，缓冲区之类的，全都由代理来搞定。举个例子，如果是串行设备，CPU 就同串行接口通信，把数据发给它后，数据再经由串行接口发给串行设备，串行设备有了反馈后，把数据发送给串行接口，再经串行接口返回给 CPU，并行设备也是如此。

任何不兼容的问题，都可以通过增加一“层”来解决。在 CPU 和外设之间的这一层就是 IO 接口。IO 接口形式不限，它可以是个电路板，也可以是块芯片，甚至可以是个插槽，它的作用就是在 CPU 和外设之间相互做协调转换，如 CPU 和外设速度不匹配，它就是变速箱，CPU 和外设信号不通用，它就是翻译机。

这样通过加了中间层后，工作就被划分成多个部分，每个部分都有专人负责，大家都轻松了，多好啊。

不过，说的还是有点抽象是吗？那就整点具体的，机箱里的声卡就是驱动音响设备的 IO 接口，本章介绍的显卡也同样是一种 IO 接口，它是用来驱动显示器的。也许您打开机箱后也未发现我说的声卡和显卡，那是不是就没有它们呢?当然不会，要是听不到声音看不到图像，人们买电脑干吗？用来学习的？其实它们被集成在主板芯片组中了，您用的就是集成声卡和集成显卡。这下清楚多了吧，下面咱们还是继续说点抽象的。

IO 接口是连接 CPU 与外部设备的逻辑控制部件，既然称为逻辑，就说明可分为硬件和软件两部分。硬件部分所做的都是一些实质具体的工作，其功能是协调 CPU 和外设之间的种种不匹配，如双方由于速度不匹配，那 IO 接口就实现数据缓冲以减少等待时间，数据格式不匹配，IO 接口就在这两种格式间互相转换。IO 接口内部实际上也是由软件来控制运作的，这就是所谓的“逻辑”部分，所以软件是指用来控制接口电路工作的驱动程序以及完成内部数据传输所需要的程序。

既然提到了软件，这就意味着编程，这样一来，IO 接口芯片又可按照是否可编程来分类，可分为可编程接口芯片和不可编程接口芯片。

接口的作用是连接处理器和外部设备，如果外部设备很简单，傻瓜型的，不需要设定就直接能用，就可以用不可编程接口芯片与处理器连接，不可编程接口芯片是种非常简单的 IO 接口。

当然物理设备还是很贵重的，并且计算机中的 IO 接口数量也是有限的，所以我们当然希望 IO 接口功能越多越好，可以设置多种工作模式，甚至允许多个外部设备通过同一个 IO 接口芯片与处理器连接。计算机与 IO 接口的通信是通过计算机指令实现的，当我们需要定制某些功能时，我们也必须用计算机指令告诉 IO 接口：哪些设备连接在此 IO 接口上，此 IO 接口的工作模式等。这种通过软件指令选择 IO 接口上的功能、工作模式的做法，称为“IO 接口控制编程”。这通常是用端口读写指令 in/out 来实现的，后面会说到。

CPU 太忙了，它的时间特别宝贵，为了简化 CPU 访问外部设备的工作，能够轻松地同任何硬件通信，大家就约定好 IO 接口的功能。

1．设置数据缓冲，解决 CPU 与外设的速度不匹配

CPU 和外设速度上的差异可以通过设置缓冲区来解决，也就是说，数据先存储在缓冲区里，等需要的时候（无论缓冲区是否满了）就传送出去。

2．设置信号电平转换电路

CPU 和外设的信号电平不同，如 CPU 所用的信号是 TTL 电平，而外设大多数是机电设备，故不能使用 TTL 电平驱动，可以在接口电路中设置电平转换电路来解决。

3．设置数据格式转换

外设是多种多样的，输出的信息可能是数字信号、模拟信号等，而 CPU 只能处理数字信号。数字信号需要经过数/模转换（D/A）成模拟量才能被送到外设以驱动硬件，模拟量也同样需要经过模/数（A/D）

转换成数字量才能被 CPU 处理。所以接口电路中需要包括 A/D 转换器和 D/A 转换器。另外，即使双方使用的都是数字信号，这也牵涉到格式和字长的问题，如 CPU 使用的是 8 位、16 位或 32 位并行数据，而外设使用并行或串行数据都有可能，所以 IO 接口中必须能够识别格式并且转换成对方需要的形式才行。

4．设置时序控制电路来同步 CPU 和外部设备

硬件的工作也按照某种时序，它们都有自己的时序系统，就像 CPU 工作在自己的晶振时序上一样。双方时序不同，接口电路就要协调这两种不同的时间计法。例如，CPU 发控制信号、定时信号给 IO 接口电路，IO 接口用它们来控制和管理硬件。随后硬件有了反馈后，其应答信号也需要通过接口返回给 CPU，这样 CPU 先“问”，硬件后“回答”，就实现了一次握手，之后便可以实现 IO 的同步操作。

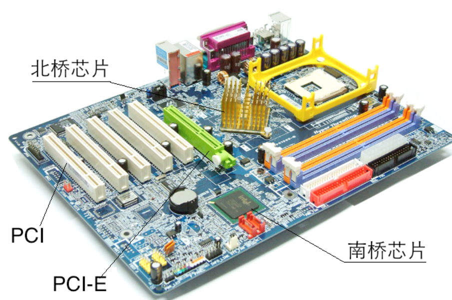
5．提供地址译码

CPU 同多个硬件打交道，每个硬件要反馈的信息很多，所以一个 IO 接口必须包含多个端口，即 IO 接口上的寄存器，来存储这些信息内容。但同一时刻，只能有一个端口和 CPU 数据交换，这就需要 IO 接口提供地址译码电路，使 CPU 可以选中某个端口，使其可以访问数据总线。

在后来新加入的硬件只要符合此约定就能同 CPU 数据交换，这样 CPU 就可以轻松应对种类万千的硬件啦。

既然都说到 IO 接口了，不知道各位有没有疑问，CPU 是怎样访问到 IO 接口呢？肯定得有个链路吧？什么？有隐约听到有同学开玩笑说：CPU 用无线访问其他设备。哈哈，不知道各位听说过没有，无线的终端是有线。无论无线设备再如何强大，最终在网络的那一头，必然是物理设备在支撑。所以，不如用个物理链路啦，你看主板上密密麻麻的线路就知道为什么有些人称主板为线路板啦。其实这物理链路就是一组电线，这条电线用于传送信号，故称为信号线。那些想和其他硬件交流的设备就要想办法连接到这条线上。由于这条电线是供大家使用的公共线路，属于所有设备共享，所以形象地称之为 bus，公共汽车，也就是大家所说的总线。总之，总线并不是抽象出来的东西，它就是把大家连接到一起的电线。再形象一点说，总线就像一条高速公路，这公路上有很多出口可以让汽车进出，汽车在计算机中就相当于各种硬件设备，可以选择连接上总线，也可以选择从总线分离。

同一时刻，CPU 只能和一个 IO 接口通信，当很多的 IO 接口同时想和 CPU 对话时，面对众多接口的爱慕，CPU 会选择和谁单独一叙呢？这个工作不应该由 CPU 来做，前面说过啦，CPU 太忙了，还是好钢使在刀刃上吧，既然分层能解决问题，咱们再加一层，这一层的责任是除了仲裁 IO 接口的竞争，还要连接各种内部总线。由于它的使命，它的名字就叫做输入输出控制中心（I/O control hub，ICH），也就是南桥芯片，如图 3-11 所示。



▲图 3-11　主板上的南桥

说到了南桥，多少还要提一句北桥芯片的作用。图 3-11 上标上北桥部分其实是散热片，在它下面才是北桥。由于南桥和北桥一般是成对出现的，至少在支持 Intel CPU 的主板中是这样的（话说，AMD 为了减少 CPU 同北桥交换数据的成本，已经把北桥的工作放到了 CPU 内部，所以支持 AMD 的板子上未必有北桥芯片）。南桥用于连接 pci、pci-express、AGP 等低速设备，北桥用于连接高速设备，如内存。

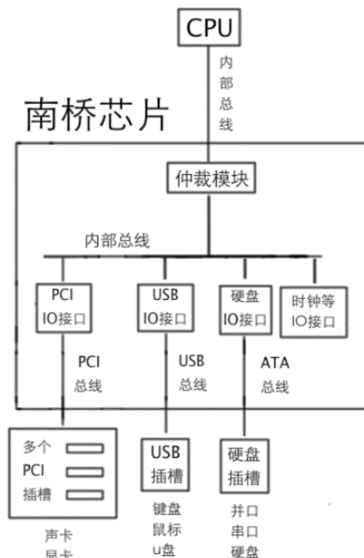
好啦，说完啦，点到为止，还得继续说南桥。CPU 通过内部总线连接到南桥芯片中的内部，这个内部总线是专用的，它只通向位于南桥中的 CPU 接口。说的好炫，其实还是一条电线而已。从名字上可以看出，南桥二字中的桥，其实对应的是 hub 这个单词，它们都意为“公共、集合”，所以不难想像，在南桥内部集成了一些 IO 接口，如并口硬盘 PATA（就是我们平时所说的 IDE 硬盘）、串口硬盘 SATA、USB、PCI 设备、电源管理等接口。由于这些接口对微型计算机来说必不可少，它们就直接扎根在南桥内部啦。

在南桥内部的接口对微型计算机来说是不可少的，除了这些之外，那些可有可无的设备，难道南桥就不管了吗？必须得管，这毕竟是它的工作。南桥芯片内部总线示意如图 3-12 所示。

为了支持这些非必要的设备（当然主要是为了方便扩展，不易扩展的产品意味着从出生那天就开始走向灭亡），南桥提供了专门用于扩展的接口，这就是 PCI 接口。在主板上有很多插槽，它们就是预留的 pci 接口，pci 设备可以即插即用。由于它们延伸到了南桥外面，又像公路一样，很多 pci 设备都可以连接上来，

所以这条延长的 PCI 接口便成了 PCI 总线。结合图 3-11 和图 3-12。看到主板上那些并排的插槽时，大家要想到，它们其实都“骑”在一条电线上，这样理解总线容易些吗？

看到一个总线又一个总线的，如果感觉烦乱的话，说明总线这个词意义不明确，也许翻译成“公共线路”最直接。简单的东西通过术语搞得好深奥，完全一副不明觉厉的样子。其实大家想想看，这么多设备要实现相互通信，不得用电线连接到一起吗？只不过大多数情况下，连接到这条电线上的设备不止两个，总的数量较多，所以称之为“总”线。由于用途不同，这些电线有了各种各样的名字，如地址总线、数据总线、ISA 总线等。总之，不要被总线这个词吓到，它其实就是电线。



▲图 3-12　南桥芯片内部总线示意图

以上都是接口的硬件部分，咱们最终是要通过软件方式使用这些硬件，下面看看咱们为了驱动这些硬件要做什么。

IO 接口在诞生之初，就被设计成要通过寄存器的方式同 CPU 通信，其内部有专用于数据交互的寄存器，只不过这里所说的这些寄存器位于 IO 接口中，为了区别于 CPU 内部的寄存器，IO 接口中的寄存器就称为端口（这可不是网络应用程序所开的那种端口，如网络服务器会启动 80 端口，这是两码事）。

IO 接口是连接 CPU 和硬件的桥梁，一端是 CPU，另一端是硬件。端口是 IO 接口开放给 CPU 的接口，一般的 IO 接口都有一组端口，每个端口都有自己的用途，甚至有时，一个端口在不同情况下有不同的用途。可见 IO 接口另一端的硬件，相对来说还是很复杂的，所以，当看到某个 IO 接口上那么多端口的介绍而烦乱时，停止抱怨，IO 接口已经为咱们极大地简化了操作。

端口也是寄存器，寄存器就有数据宽度，有 8 位、16 位、32 位，各个设备是不一样的，看厂商自己安排了。

如何访问到端口呢？外设中的 rom 既然可以通过内存映射来访问，端口也可以，确实有些微机系统中是这样做的，把一些内存地址作为端口的映射，访问这些内存地址就相当于访问了这些端口。还有一些微机系统把端口独立编址，把所有端口从 0 开始编号，位于一个 IO 接口上的所有端口号都是连续的。以后讲解硬盘的时候大家就会看到了。

IA32 体系系统中，因为用于存储端口号的寄存器是 16 位的，所以最大有 65536 个端口，即 0～65535。

要是通过内存映射，端口就可以用 mov 指令来操作。但由于用的是独立编址，所以就不能把它当作内存来操作，因此 CPU 提供了专门的指令来干这事，in 和 out。

Intel 汇编语言的形式是：操作码目的操作数，源操作数。Intel 采用这种格式的原因可能是觉得这样表达“目的操作数”=“源操作数”更形象，如同 a=6 这种形式。

in 指令用于从端口中读取数据，其一般形式是：

（1）in al, dx；

（2）in ax, dx。

其中 al 和 ax 用来存储从端口获取的数据，dx 是指端口号。

这是固定用法，只要用 in 指令，源操作数（端口号）必须是 dx，而目的操作数是用 al，还是 ax，取决于 dx 端口指代的寄存器是 8 位宽度，还是 16 位宽度。

out 指令用于往端口中写数据，其一般形式是：

（1）out dx, al；

（2）out dx,ax；

（3）out 立即数, al；

（4）out 立即数, ax。

注意啦，这和 in 指令相反，in 指令的源操作数是端口号，而 out 指令中的目的操作数是端口号。

如果上面看着有点凌乱，给大家总结一下 in 和 out 指令共性。

（1）在以上两个指令中，dx 只做端口号之用，无论其是源操作数或目的操作数。

（2）in 指令从端口读数据，可以认为端口是数据源，所以端口出现在“源操作数”的位置。读出来的数据要有个“目的地”来存放，所以 in 指令中存放数据的地方出现在“目的操作数”位置。

out 指令是把数据写入端口指向的寄存器，在这里，端口是数据的“目的地”，所以端口出现在目的操作数的位置。待写入的数据总该有个“来源”，所以 out 指令中的“源操作数”是数据来源。

在以上的两个指令中，端口号和数据的位置，取决于它们各自的角色是源操作数，还是目的操作数。

（3）在以上两个指令的两个操作数中，无论是对于源操作数，还是目的操作数，除端口号外，那个作为数据的操作数（in 指令中作为数据目的地，out 指令中作为数据源），一律用 al 寄存器存储 8 位宽度的数据，用 ax 寄存器存储 16 位宽度的数据，至于用 al，还是 ax 存数据，要看端口指向的寄存器宽度是多少，它要和端口寄存器的位宽保持一致，不能丢失数据精度。

（4）in 指令中，端口号只能用 dx 寄存器。

（5）out 指令中，可以选用 dx 寄存器或立即数充当端口号。

真心希望大家看完后不会更乱了。

好啦，有了这些硬件相关的知识，对以后我们操作其他硬件来说足够了，我们不需要学习得多全，够用就好。还是那句话，以后用到哪再学不迟。大家辛苦了，祝大家今后学习顺利。

3.3.2 显卡概述

在上一章的 mbr 中我们刚刚向屏幕输出了“1 MBR”这几个字符，这种喜悦还没有过去，我就要给大家泼冷水了：这种打印字符的方法马上就用不了啦。

mbr 运行在实模式下，所以在实模式下也可以用 BIOS 的 0x10 中断打印字符串，这是因为：首先中断向量表只在实模式下存在，BIOS 中断是要依赖于中断向量表的。可是，将来的世界是由保护模式罩着的，保护模式下就没有中断向量表了，所以也就无法用 BIOS 中断。其次，不希望有更多的依赖，好不容易脱离了对操作系统的依赖，又引入了一个新的依赖，这不科学。最后，难道大家不想直接同显卡说几句话吗？

万变不离其宗，肯定的是 BIOS 的中断例程中凡是涉及向屏幕打印之类的功能，必然也是通过操作显卡来实现的，只是通过封装成中断处理程序给大家方便调用而已，我们也不用关心显卡操作的细节。等下，往屏幕上输出信息操作的对象不是显示器吗？你这一直说显卡是怎么回事？如果您也有这样的疑问，我这捎带着说解释一下。

某些 IO 接口也叫适配器，适配器是驱动某一外部设备的功能模块。显卡也称为显示适配器，不过归根结底它就是 IO 接口，专门用来连接 CPU 和显示器。我们想操作显示器，没有直接的办法，只能通过它的 IO 接口——显卡。

稍微说一下显卡。自从几年前 AMD 把 ATI 收购之后，市面上的显卡就分为两大类了，A 卡和 N 卡。A 卡是指以 AMD 为阵营的显卡厂商，N 卡是以 nvidia 为阵营的显卡厂商。大家平时见到的七彩虹、技嘉、昂达之类的显卡，它们用的核心要么是 A 卡，要么是 N 卡，有的厂商两个核心都用，开发各自的版本。他们不自己研发 GPU（显卡的 CPU 称为 GPU），只是在人家的基础上做本地化开发。这种关系就像安卓手机和安卓原生系统一样。

话说我在 2003 年的时候见过一块特别霸气的显卡，这块显卡一看就是发烧级的。为什么呢？一般的显卡是要插在主板上的，由于这块显卡做得特别大，看上去感觉像是主板插在了显卡上。

显卡是 pci 设备，所以是安装在主板上 pci 插槽上的，pci 总线是共享并行架构，并行数据就要保证数据发送后必须同时到达目的地，因为这关系到数据的顺序，不能发过去后成一团乱麻。例如 8 位并行总线就需要同时发送这 8 位，接收方也要同时接收这 8 位才行。虽然貌似并行传输是高效的，但对于要保证同时接收 n 位数据，这是有困难的，随着并行数据的位宽越来越大，这种困难也越来越明显。于是串行传输很好地解决了这一问题，一次只发一位，这样顺序问题解决了，数据到目的地看再组合到一起就成了。于是就有了 PCI Express 总线，这就是串行设备，简称 pcie。现在的显卡都是串口的了，包括上面说的 A 卡和 N 卡。有同学会问吧，一次一位地传输，那多慢啊，听上去不如并行传输快。但大家不要忘记了，传

输速度一部分取决于并行的数据量，一部分还要取决于传输频率呢。串口显卡一次虽然只传输 1 位，但人家传输的频率快啊，不光是显卡，现在的硬盘都是串口的，可见串行传输速率可是极高的。

总之以后我们的输出都是通过直接操作显卡来实现的，而显卡给我们的输入接口是显存和端口，我们主要用的是显存。显存作为接口，说白了，就是它把显存直接给我们用，说："把你要输出的内容写到这里面，我照着往屏幕上打印"。

好啦，本节到这儿结束了。

3.3.3　显存、显卡、显示器

为了能够看到图像，我们需要显示器。无论是哪种显示器，它都是由显卡来控制的，我们没必要了解液晶显示器和普通 CRT 显示器的差别。无底是哪种显卡，它提供给我们的可编程接口都是一样的：IO 端口和显存。

显存是由显卡提供的，它是位于显卡内部的一块内存，所以它称为显存。关注过显卡产品的同学可能会知道，有的标明了 DDR 512M，有的则声称是 DDR2 1G。这指的就是显存大小。显卡的工作就是不断地读取这块内存，随后将其内容发送到显示器。

我们能在显示器上见到的各种色彩斑斓的图像，说明显卡可以让显示器工作在图形模式，能够在显示器上看到 Linux 终端上的黑屏白字，说明显卡可以让显示器工作在字符模式。屏幕是由密密麻麻的像素组成的，显存中的每一位都对应屏幕上的一个像素点。

我们打开一个网页后，里面所加载的图片，就是显示器在图形模式下的效果。按理说，显存中的一位对应一个像素，该位要么是 0，要么是 1，如果让它显示颜色，一个像素顶多显示黑白两色啊，它是如何显示彩色的呢？是啊，一位只能显示两种颜色，看来只有增加位数来达到彩色的效果了。各位肯定听说过 24 位真彩色吧，没听过也没关系，就当您听过了，哈哈，其实 24 位真彩色就是用 24 个 bit 表示一个颜色，也就是 3 字节的数据量来表示一种颜色。能表示多少种颜色呢？2 的 24 次方等于 16777216 种。天啊，我平时就知道赤橙黄绿青蓝紫 7 种颜色，我不是色盲，不过这么多颜色让我分辨清楚，臣妾做不到啊。

之前有不少同学的理解只是概念性的，现在要应用到实践中啦，在黑白图形模式中，显存位与屏幕像素是 1 对 1 的，因为只有两种颜色，所以只要显存中的对应位置为 1，屏幕上的相应像素就被点亮，呈现的是白色。若该位为 0，该像素就不会被点亮，只要不管该像素就是黑色，所以用黑色壁纸当桌面，才真正是在物理上保护了显示器。而在真彩色中，是用 24 位对应一个像素，所以才呈现出彩色。

显示器分不清楚给它的数据是文本，还是图像，在它眼里全都是图像，粒度更细致点来说，全是像素信息，即像素的位置及像素的颜色。只有人才能分得出这是文字，那是花草，那是星空。所以，对于图像的输出，最直观的想法是：人们想输出什么图像就是计算出要将哪些像素点亮。这简短的一句话，有没有让您心中仿佛有一万只草泥马奔腾而过？什么，没有？那您帮我输出爱因斯坦的肖像给我看，注意，我要看清他的头发。现在草泥马是两万只了？听上去这种用像素拼凑图像的方法真的不亚于愚公移山。

计算机的发明是为了解决问题，而不是带来问题，聪明的工程师当然有更人道的方法，解决问题的方式是一个字符对应一字节的编码，只要往显存中写入这个编码，显卡就知道这是要打印此字符，由它帮你完成像素的拼凑。比如字符 A 的编码就是 0x41，在它后面的字符 B 的编码增加 1，即 0x42。

编码本质上就是按照某种约定生成一组数据，这种约定可以是某种数学关系，如算法、公式，或者是某种固定关系，如像"藏头诗"这类，关键字是文本中固定的位置，或者将这种对应关系事先写到表格中，通过查表得到输出。解码就是根据此约定来做逆运算。破译就是找出编码中使用的是哪种约定并进行解码。

最常见的编码就是交警指挥交通时的手势，每种手势的意义司机都清楚，但如果不懂交通规则，自然就不明白了。

这样，大家都约定好了，以后字符 A 就用十六进制数字 0x41 来表示，甭管是谁发来的这个数，我就认为这是字符 A。当然这还是有应用的前提，也得分场合，不是说只要 0x41 就是字符 A，应该说是接收端把接收的内容当作文本来处理时，0x41 才被赋予字符 A 的意义。数字的意义是被生产者和消费者共同定义的，主要是看处理双方如何看待这一组数字，这就是约定的体现。

既然是约定，大家都要共同遵守才行，不能我发 0x38 代表 A，你认为 0x38 是 delete，坚决不能另起山头自立门户。所以为了大家都有据可依，一套字符编码横空出世，这就是名气响当当的 ASCII 码。

美国信息互换标准代码（American Standard Code for Information Interchange，ASCII）。它是由美国国家标准学会（American National Standard Institute，ANSI）制定的，是标准单字节字符编码方案，用于描述纯文本。标准 ASCII 码也叫基本 ASCII 码，用 7 位二进制数来表示大、小写字母，数字 0～9、标点符号以及一些控制字符。标准 ASCII 表中的字符分为两大类，一类是不可见字符，控制字符属于此类，其余为可见字符。下面按这两种分类附上标准 ASCII 码表，见表 3-13、表 3-14。

表 3-13　　ASCII 码中的控制字符

ASCII 控制字符			
十进制	十六进制	缩　写	名称/意义
0	00	NUL(null)	空字符（Null）
1	01	SOH(start of headling)	标题开始
2	02	STX (start of text)	本文开始
3	03	ETX (end of text)	本文结束
4	04	EOT (end of transmission)	传输结束
5	05	ENQ (enquiry)	请求
6	06	ACK (acknowledge)	确认回应
7	07	BEL (bell)	响铃
8	08	BS (backspace)	退格
9	09	HT (horizontal tab)	水平定位符号
10	0A	LF (NL line feed, new line)	换行
11	0B	VT (vertical tab)	垂直定位符号
12	0C	FF (NP form feed, new page)	换页
13	0D	CR (carriage return)	回车
14	0E	SO (shift out)	取消变换（Shift out）
15	0F	SI (shift in)	启用变换（Shift in）
16	10	DLE (data link escape)	跳出数据通讯
17	11	DC1 (device control 1)	设备控制一（XON　启用软件速度控制）
18	12	DC2 (device control 2)	设备控制二
19	13	DC3 (device control 3)	设备控制三（XOFF　停用软件速度控制）
20	14	DC4 (device control 4)	设备控制四
21	15	NAK (negative acknowledge)	拒绝接收
22	16	SYN (synchronous idle)	同步用暂停
23	17	ETB (end of trans. block)	区块传输结束
24	18	CAN (cancel)	取消
25	19	EM (end of medium)	连接介质中断
26	1A	SUB (substitute)	替换
27	1B	ESC (escape)	跳出
28	1C	FS (file separator)	文件分割符
29	1D	GS (group separator)	组群分隔符
30	1E	RS (record separator)	记录分隔符
31	1F	US (unit separator)	单元分隔符
127	7F	DEL(delete)	删除

表 3-14　　ASCII 码中的可见字符

ASCII 码可见字符								
十进制	十六进制	图形	十进制	十六进制	图形	十进制	十六进制	图形
32	20	（空格）	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	（	72	48	H	104	68	h
41	29	）	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	，	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	：	90	5A	Z	122	7A	z
59	3B	；	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	～
63	3F	?	95	5F	_			

有了这套标准，任何字处理软件只要认真遵守，就能得到别人的理解和认可。不知您想过没有，在我们人类看了 ASCII 这套标准后，我们已经变成了字处理软件，我们要想往显示器或任何一个文本处理系统中输出文本信息，我们也得必须按照这套规则来编码了。于是乎，我们往屏幕上输出字符 A，我们要输出数字 0x41。输出字符 a，我们输出数字 0x61。那我想往屏幕上输出 0，直接输出数字 0 能行吗？由于 ASCII 是一套“字符”标准，它只会打印出字符，数字 0 可不是在屏幕上输出的 0，屏幕上的 0 那可是字符'0'，所以想输出 0，得输出数字 0x30。

介绍了这么多，现在就差体验一把了，之前说过了，显存是显卡给我们的接口，咱们得往显存里写点东西才行，可是显存在哪里，怎样写？

在表 1-1 中，我们介绍了内存的布局。从中我把与显存相关的部分列在了下面，大家看下显卡各种模式的内存分布，见表 3-15。

表 3-15　　显存地址分布

起　　始	结　　束	大　　小	用　　途
C0000	C7FFF	32KB	显示适配器 BIOS
B8000	BFFFF	32KB	用于文本模式显示适配器
B0000	B7FFF	32KB	用于黑白显示适配器
A0000	AFFFF	64KB	用于彩色显示适配器

各外部设备都是通过软件指令的形式与上层接口通信的，显卡（显示适配器）也不例外，所以它也有自己的 BIOS。位置是 0xC0000 到 0xC7FFF。显卡支持三种模式，文本模式、黑白图形模式、彩色图形模式。我们只关注文本模式就好了，最终我们要实现类似 Linux 终端那样的字符界面。

我们平时看的电影，一秒 24 帧，每一帧都是一幅图片，有的还是高清电影，可想而知，1 秒的数据量也是很大的。为了提速，避免视觉上产生延时，硬件系统干脆让我们直接和显卡接触，免得数据再经由第三道手而影响效率，直接把数据往显存中填就好了。

之前和大家介绍过了，地址总线的范围不只是主板上插的内存条的容量，内存条只是地址总线所能达到的范围中的一小部分。指令中所需的任何一个地址，都是地址总线帮咱们去寻址的。地址只是个数字，地址总线把此数字指向哪个存储介质，此地址就落到哪个介质上的某个存储单元中。地址指向哪里，最终是地址总线说了算。如果有同学误以为访问某个 rom 的地址，是先访问到我们的内存条后，再由计算机内的某种神奇力量将其映射到该 rom，这就不对了。

从起始地址 0xB8000 到 0xBFFFF，这片 32KB 大小的内存区域是用于文本显示。我们往 0xB8000 处输出的字符直接会落到显存中，显存中有了数据，自然显卡就将其搬到显示器屏幕上了，这后续的事情咱们是不需要处理的，咱们只要保证写进显存的数据是正确的就可以。

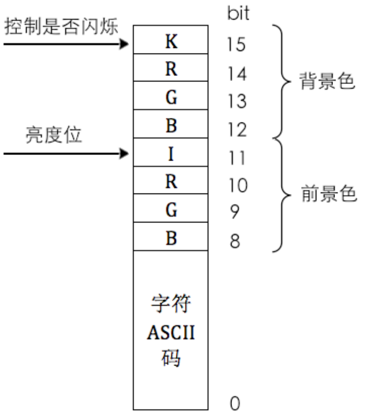
屏幕上可以显示多少个字符呢？这要取决于要用哪种文本模式了。

显卡的文本模式也是分为多种模式的，用“列数*行数”来表示，如 80*25，40*25，80*43 或者 80*50，它们的乘积是整个屏幕上可以容纳的字符数。不同的模式可容纳的字符数不同，如 80*25 表示一行 80 个字符，共 25 行。显卡在加电后，默认就置为模式 80*25，也就是一屏可以打印 2000 个字符。我们也在这个默认模式下工作了。

即使在文本模式下，也可以打印出彩色字符。可是 ASCII 码都是 1 字节大小，即使标准 ASCII 码也要用 7 位来为一个字符编码，只剩下那 1 位顶多只能表示黑白两种颜色。聪明的你肯定想到啦，必然是用一个字节来表示字符本身，再用另外的字节来表示其属性。答对啦，事实也是如此。每个字符在屏幕上都是由 2 个字节来表示的，而且是连续的 2 个字节。

说到这里我们先算一笔账，显存是从 0xB8000 到 0xBFFFF，范围是 32KB，一屏可以显示 2000 个字符，显示器上的每个字符占 2 字节大小，故每屏字符实际占用 4000 字节。这样，我们的 32KB 的显存可以容纳 32KB/4000B 约等于 8 屏的数据。所以您懂了为什么 Linux 可以用 alt + Fn 键实现 tty 的切换，当然这只是原理，具体的实现要涉及到显卡的寄存器设置。显卡上的寄存器还是非常多的，我怕此时将它们列出来会打击大家学习积极性，以后需要时再说吧。

屏幕上每个字符的低字节是字符的 ASCII 码，高字节是字符属性元信息。在高字节中，低 4 位是字符前景色，高 4 位是字符的背景色。颜色用 RGB 红绿蓝三种基色调和，第 4 位用来控制亮度，若置 1 则呈高亮，若为 0 则为一般正常亮度值。第 7 位用来控制字符是否闪烁（不是背景闪烁）。这两字节如图 3-13 所示。



▲图 3-13　字符及其属性

大家知道，用 R 红色、G 绿色、B 蓝色这三种颜色以任意比例混合，可以搭配出其他颜色，其他颜色被认为都可以由这三种颜色组合而成。不过由于在文本模式下的颜色极其有限，RGB 的各部分比例要

么是 1（全部），要么是 0（没有），所以其组合出的颜色屈指可数，为了让大家测试字符颜色更加方便，给大家提供这三种颜色的组合，见表 3-16。

表 3-16　文本模式中字符颜色

R	G	B	颜　色	
			I=0	I=1
0	0	0	黑	灰
0	0	1	蓝	浅蓝
0	1	0	绿	浅绿
0	1	1	青	浅青
1	0	0	红	浅红
1	0	1	品红	浅品红
1	1	0	棕	黄
1	1	1	白	亮白

从上面可以看出只要亮度位 I 置 1，颜色就是变亮变浅。大家可以结合 K 位来测试上面的颜色。

3.3.4　改进 MBR，直接操作显卡

到目前为止，说了一部分有关显存的内容，这对于一般的输出来说已经足够了，下面咱们可以尝试写显存啦。我们将之前的 MBR 改造一下，保留滚屏的操作，只修改有关输出的部分，即把通过 BIOS 的输出改为通过显存，你会发现，其实反而更容易，请见代码 3-4。

代码 3-4　（project/c3/a/boot/mbr.S）

```
 1 ;主引导程序
 2 ;
 3 ;LOADER_BASE_ADDR equ 0xA000
 4 ;LOADER_START_SECTOR equ 0x2
 5 ;------------------------------------------------------------
 6 SECTION MBR vstart=0x7c00
 7    mov ax,cs
 8    mov ds,ax
 9    mov es,ax
10    mov ss,ax
11    mov fs,ax
12    mov sp,0x7c00
13    mov ax,0xb800
14    mov gs,ax
15
16 ;清屏
17 ;利用 0x06 号功能，上卷全部行，则可清屏
18 ; -----------------------------------------------------------
19 ;INT 0x10   功能号：0x06    功能描述：上卷窗口
20 ;------------------------------------------------------
21 ;输入：
22 ;AH 功能号= 0x06
23 ;AL = 上卷的行数（如果为 0，表示全部）
24 ;BH = 上卷行属性
25 s;(CL,CH) = 窗口左上角的(X,Y)位置
26 ;(DL,DH) = 窗口右下角的(X,Y)位置
27 ;无返回值:
28    mov     ax, 0600h
29    mov     bx, 0700h
30    mov     cx, 0             ; 左上角: (0, 0)
31    mov     dx, 184fh          ; 右下角: (80,25),
32                      ; VGA 文本模式中，一行只能容纳 80 个字符，共 25 行
33                      ; 下标从 0 开始，所以 0x18=24，0x4f=79
34    int     10h               ; int 10h
35
36     ; 输出背景色绿色，前景色红色，并且跳动的字符串"1 MBR"
```

```
37    mov byte [gs:0x00],'1'
38    mov byte [gs:0x01],0xA4   ; A表示绿色背景闪烁，4表示前景色为红色
39
40    mov byte [gs:0x02],' '
41    mov byte [gs:0x03],0xA4
42
43    mov byte [gs:0x04],'M'
44    mov byte [gs:0x05],0xA4
45
46    mov byte [gs:0x06],'B'
47    mov byte [gs:0x07],0xA4
48
49    mov byte [gs:0x08],'R'
50    mov byte [gs:0x09],0xA4
51
52    jmp $                    ; 通过死循环使程序悬停在此
53
54    times 510-($-$$) db 0
55    db 0x55,0xaa
```

前 36 行除第 13～14 行以外，和上一版本的 MBR 一样，忘记的话也不用翻回去看了，直接看注释简单了解一下就好，剧透一下，以后连滚屏我们都要直接通过显卡来搞定。

前面说过了，显存文本模式中，其内存地址是 0xb8000，忘记的话可以往前翻翻“表 3-15”显存地址分布。时刻要清楚，我们目前是在实模式下编程，实模式下内存分段访问策略是“段基址*16+段内偏移地址”。注意，要考虑到最终地址的段基址要乘以 16，所以咱们选择的段基址必须是除以 16 以后的值。目标地址是 0xb8000，按照以上策略，有多种“段基址+段内偏移地址”的组合可以拼凑出此地址。最直观的段基址为 0xb800，即 0xb8000 除以 16，也就是右移 4 位，偏移地址为 0。

所以第 13 行和第 14 行往 gs 寄存器中存入段基址。这里和大家说明一下，显存段基址放在哪个寄存器中都是没关系的，对于访问的是数据来说，如果不用 ds 做段基址寄存器，就要在寻址中“显式地”指明要用哪个段寄存器的值作为段基址。这个“显式的”的段寄存器叫作段跨越前缀，有的书中叫段超越前缀，个人觉得意义不明确。何为超越？由于有“跨段访问”的说法，所以咱们这里统一为段跨越前缀。“段跨越”相对好理解，如 CPU 的访存策略是“段地址+段内偏移地址”。堆栈段的寄存器是 SS，代码段寄存器 CS，这两者不存在默不默认之说，因为它们都不能改变。不过对于数据段来说却有些不同，默认的寄存器是 DS，但其是可以改变的。一般访问数据时只要给出偏移地址就可以了，这是因为已经存在了默认的段寄存器 DS，所以访存中给出的偏移地址便是相对于 DS 的偏移量，也就是说访问的地址属于以 DS 为起始的段（是指一般意义上的分段机制，不考虑实模式或保护模式）。但若不想用这个段了，或者访问的地址不属于这个段，想“跨越”这个默认段，而用新的段基址，“跨过”DS 的限制，这就是“跨越”的理解。而“前缀”的意思是在编译后的机器码中，指定的这个新的段寄存器会出现在 IA32 指令格式中的“前缀”字段，可以参见表 3-1“IA32 指令格式”。基于以上两点，为代替默认段基址寄存器而改用的新的段基址寄存器，称为段跨越前缀。

我们在第 37～50 行执行的 mov 操作都是往显存中写字符。拿 37 行和 38 行举例，第 37 行的“mov byte [gs：0x00]，'1'”，是往以 gs 为数据段基址，以 0 为偏移地址的内存中写入字符 1 的 ASCII 码。按之前我们讲过的，写入 1 时，要写入 1 的 ASCII 码 0x31。这是最直接的做法。但编译器诞生的意义就是为了给大家带来方便，尽管我们可以把 37 行的代码改为 mov byte [gs：0x00]，0x31，但这样毕竟还要自己查 ASCII 码表。编译器对于出现在代码中的字符，它会自动将其改为相应的 ASCII 码，免去了人工查表的过程。即使把表整个背下来了，本质上也是在脑子中经过了一次查表。所以，对于字符的输出，直接写出相应字符就行了，稍微有点人性化的编译器都会自动完成字符到编码的转换。

这里还有一个关键字 byte，用于指定操作数所占的空间。同类的关键字还有 word、dword 等。这些关键字指明了操作数的数据宽度（字节数），同 C 语言中的变量类型一个道理，都是指明数据所需要的存储空间。“mov byte [gs:0x00],'1'”表示的意思是：把字符 1 的 ASCII 码写入以 gs：0x00 为起始，大小为 1 字节的内存中。word、dword 分别表示 2 字节和 4 字节，意义同理。如果源操作数或目的操作数已经明确了数据宽度，在指令中就不必“显示地”指明操作数所占的空间大小了。例如 mov ax, 0x10，目的操作数 ax 是 16 位的，所以不用“显示地”在 ax 前或 0x10 前加个关键字 word。在我们的代码“mov byte [gs:0x00],'1'”

中，由于这里的'1'对应的 ASCII 码是 0x31，这是个立即数，对于立即数是无法判断它的存储空间的。它是占 1 字节，还是 2 字节？将来会不会超过 255？它现在是 0x31，还是 0x0031？这谁知道呢。可 CPU 需要知道这个 0x31 要用多少字节来存储，因为它不确定这个数将来会不会超过 255。要是用一字节来存储 0x31，万一哪天往此处存个大于 255 的数，这一字节是万万不能胜任的。

我们之前说过了，一个字符用 2 个字节来表示。低字节是字符的 ASCII 码，这里的偏移地址还是 0x00，gs 在程序开头被赋值为 0xb800，故最终地址是 0xb8000，即显存的第 0 个字节。这表示字符 1 会在屏幕的左上角。

字符的高字节是属性，所以我们在第 38 行用“mov byte [gs:0x01],0xA4”为字符添加颜色。这里的偏移地址已经变成了 0x01，是该字符'1'的高位，写入的属性值是 0xA4，这表示 K 位为 1，结合表 3-16 可知，其为红色跳动字符，绿色背景。

第 39～50 行分别在显存中创建字符'M'，'B'，'R'及其属性，拼接字符串“MBR”，原理同上。

第 52 行还是个死循环，程序会卡在这里不动。其余代码同之前一样，是为了凑足 512 字节并写入魔数 0xaa55。

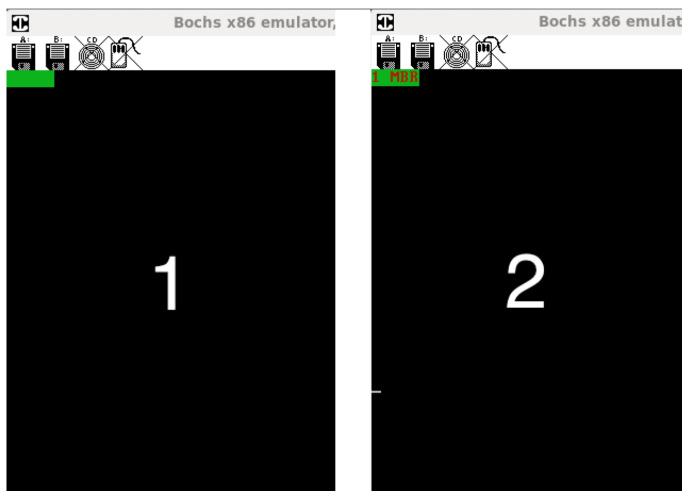
代码不多，分析到此为止，事不宜迟，立即编译。

```
nasm -o mbr.bin mbr.S 回车
```

下面将生成的 mbr.bin 写入我们的虚拟硬盘，还是用 dd 命令。

```
dd if=./mbr.bin \
of=/your_path/bochs/hd60M.img \
bs=512 count=1  conv=notrunc 回车
```

好了，按照前面介绍的方法启动 bochs，执行 c 命令，将会在屏幕的左上角出现绿色背景、红色跳动的字符。效果如图 3-14 所示。



▲图 3-14　屏幕字符闪烁

为了表示文字确实是在闪烁，所以我分别截了两张图。左半部分标注为 1 的部分是字符闪没的截图，右图标注为 2 的部分是字符再次出现的截图。

关于显卡另外的接口——端口，我们在之后用到的时候再细说，大家在此不要惦记了。

3.4 bochs 调试方法

前面第 1 章中在给大家介绍 bochs 的时候，我犹豫要不要把 bochs 调试方法也一块放进去，放在那里似乎更显得“规矩”。但总觉得放在那了大家当时也用不上，用的时候还得往回翻那么多，所以经过考虑，我决定在本章完成 MBR 的改进后再介绍 bochs 用法，这样大家能“实习”一把，练习用 bochs 调试上一

节改进后的 MBR，有助于理解 bochs 用法，记忆更深刻。

3.4.1 bochs 一般用法

bochs 是一个开源 x86 虚拟机软件。在它的实现中定义了各种数据结构来模拟硬件，用软件模拟硬件缺点是速度比较慢，毕竟全是软件来模拟，您想，虚拟机还要在软件中模拟各种中断，能不慢吗。不过它的功能非常强大，咱们应该感激 bochs 开发人员所做的贡献，真的不能抱怨，有的用就不错了是不。其优点是可移植性强，原则上只要 gcc 支持某个平台，这个平台上就可以有 bochs，从而保证了 bochs 在各平台上的畅通无阻。由于它是虚拟机，所以支持硬件级别上的调试。

bochs 的硬件调试体现在：

（1）调试时可以查看页表、gdt、idt 等数据结构；

（2）可以查看栈中数据；

（3）可以反汇编任意内存；

（4）实模式、保护模式互相变换时提醒；

（5）中断发生时提醒。

这种在硬件级别上的调试给我们提供了更大的灵活性，以后您会发现，这种硬件调试有时候会帮我们大忙。

好在 bochs 的调试风格是参照 gdb 来设计的，这对于习惯 gdb 调试的同学无疑减少了学习成本，不熟悉 gdb 调试器的同学也不必感到沮丧，我们常用的调试命令并不多，而且 bochs 的调试方法做得很人性化，发挥一下想像力也能摸索个所以然来。本书中使用的 bochs 版本是 2.6.2，以下就此版本对 bochs 的使用做大致介绍。

闲话少说，咱们先进入 bochs，看看大概有哪些内容，如图 3-15 所示。

```
[work@localhost bochs]$ ls
bin  bochs.out  bochsrc.disk  hd60M.img  share
[work@localhost bochs]$ bin/bochs -f bochsrc.disk
```

▲图 3-15　bochs 下的文件

第一行 ls 命令后，显示的是我安装的 bochs 下的文件，bin 和 share 这两个目录是 bochs 安装时创建的，bochs.out 是 bochs 运行过程中的日志文件，它是在配置文件中指定的，而在本例中，bochs 的配置文件是 bochsrc.disk。hd60M.img 是用 bin/bximage 命令创建出来的虚拟硬盘，它也需要在 bochsrc.disk 中指定后才能使用。

第二行是启动 bochs。由于我们的配置文件并不是这三个标准名称：.bochsrc、bochsrc、bochsrc.txt，所以我们需要用-f 来指定我们的配置在哪里。其实用-f 来指定是有好处的，这样我们清晰地知道哪个才是我们的配置。

如图 3-16 所示，进入 bochs 后，我们要确定下一步做什么，由于 bochs 已经将选项[6]作为默认的行为，这里直接回车就好了。

```
00000000000i[     ] reading configuration from bochsrc.disk
00000000000e[     ] bochsrc.disk:30: 'keyboard_mapping' will be replaced by new
'keyboard' option.
------------------------------
Bochs Configuration: Main Menu
------------------------------

This is the Bochs Configuration Interface, where you can describe the
machine that you want to simulate.  Bochs has already searched for a
configuration file (typically called bochsrc.txt) and loaded it if it
could be found.  When you are satisfied with the configuration, go
ahead and start the simulation.

You can also start bochs with the -q option to skip these menus.

1. Restore factory default configuration
2. Read options from...
3. Edit options
4. Save options to...
5. Restore the Bochs state from...
6. Begin simulation
7. Quit now

Please choose one: [6]
```

▲图 3-16　bochs 启动画面

像很多提供控制台的软件一样，直接键入 help 会显示帮助信息。进入 bochs 后，键入命令 help 后回车，看看 bochs 给我们准备了什么礼物。help 命令的输出如图 3-17 所示。

简短看一下 help 信息就可以了，不要细看每个命令，很多是我们不需要的，后面我会讲解常用的部分。在这里，我本想将所有调试命令的说明先搬到这里，然后再带大家做个小例子。但根据我以往读书的经验，每次将一堆事先用不到的东西一股脑地塞过来时，我完全不感冒，因为不知道哪个重要，而且看到哪个命令都是靠想像力来琢磨这是讲的什么，甚至我每次都会翻过去直接看后面，没有经过实践的东西是不会被真正理解的。所以我决定先带着大家做个小例子，过程中大家有疑问没关系，带着问题去看后面命令的详细说明，这样效果更好。

```
Please choose one: [6]
00000000000i[     ] installing x module as the Bochs GUI
00000000000i[     ] using log file bochs.out
Next at t=0
(0) [0x0000fffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ; ea5be000
f0
<bochs:1> help
h|help - show list of debugger commands
h|help command - show short command description
-*- Debugger control -*-
    help, q|quit|exit, set, instrument, show, trace, trace-reg,
    trace-mem, u|disasm, ldsym, slist
-*- Execution control -*-
    c|cont|continue, s|step, p|n|next, modebp, vmexitbp
-*- Breakpoint management -*-
    vb|vbreak, lb|lbreak, pb|pbreak|b|break, sb, sba, blist,
    bpe, bpd, d|del|delete, watch, unwatch
-*- CPU and memory contents -*-
    x, xp, setpmem, writemem, crc, info,
    r|reg|regs|registers, fp|fpu, mmx, sse, sreg, dreg, creg,
    page, set, ptime, print-stack, ?|calc
-*- Working with bochs param tree -*-
    show "param", restore
<bochs:2>
```

▲图 3-17　bochs 帮助信息

大体上 bochs 的调试命令分为“Debugger control”类、“Execution control”类、“Breakpoint management”类、“CPU and memory contents”类。在每个大类别中的关键字都是一个调试命令，看上去也不少，每个命令的用法不同。不过好在咱们今后用的命令不多，大家不必一下子把这些全部掌握，咱们还是本着递归式学习方法，用到了再学不迟。咱们这里先只做个笼统的介绍，做个小例子，带大家入入门，今后可以自己摸索了。

图 3-17 中，根据第二行的提示，“help+命令”，可以显示命令的简短描述信息。那咱们就试一下。

在“CPU and memory contents”类中，有 x、xp 命令。这两个命令是用来查看内存的，它们的区别是 x 命令后接线性地址，xp 命令后接 physical 物理地址。在目前的实模式中，只能通过物理地址来查看内存，先看看 xp 命令是怎么用的，一会咱们用 xp 命令来做个测试。键入 help xp 回车，如图 3-18 所示。

```
<bochs:3> help xp
x  /nuf <addr> - examine memory at linear address
xp /nuf <addr> - examine memory at physical address
    nuf is a sequence of numbers (how much values to display)
    and one or more of the [mxduotcsibhwg] format specificators:
    x,d,u,o,t,c,s,i select the format of the output (they stand for
        hex, decimal, unsigned, octal, binary, char, asciiz, instr)
    b,h,w,g select the size of a data element (for byte, half-word,
        word and giant word)
    m selects an alternative output format (memory dump)
<bochs:4>
```

▲图 3-18　xp 指令帮助信息

说明一下，bochs 中用到的“字”并不是 2 字节，而是 4 字节。在图 3-17 的倒数第 4 行，提示用 b,h,w,g 来选择一个“显示单元”的大小。例如 b 是指一字节。h 是指半个字，2 个字节。w 是指一个字，4 个字节。g 是指双字，8 字节。用 xp 或 x 指令查看内容是以“显示单元”为单位，不是以字节。所以如果不指定数据单位大小，默认以 4 字节为单位来显示。例如 xp 0x7c00，将显示从 0x7c00 开始的 4 个字节。

bochs 中支持八进制、十进制、十六进制的数字。八进制按照以 0 开头的写法即可，十进制自然不用多说，对于十六进制却有点限制，只支持 0x 前缀的形式，不支持 h 后缀的形式。如：

八进制：011

十进制：11

十六进制：0x11

在咱们调试过程中最常用的还是十六进制，它的每一位直接和字节中的每 4 位对应，观察起来还是较方便一些。

继续看 xp 指令“xp /nuf <addr>”，nuf 是指一个数字序列，这是三个参数，n 用来分别指定要显示的“显示单元”数，u 指“显示单元”大小，f 是指要用哪种进制显示。最终要显示几个字节，是以“显示单元大小 u*显示单元个数 n”来决定的。addr 可以是以上三种进制的数字。即使不明白，看着有点晕也没有关系，后面我们还会对此命令细说。用法如图 3-19 所示。

```
<bochs:8> xp/b 0x7c00
[bochs]:
0x00007c00 <bogus+       0>:    0x00
<bochs:9> xp 0x0
[bochs]:
0x00000000 <bogus+       0>:    0x00
<bochs:10> xp/w 0x0
[bochs]:
0x00000000 <bogus+       0>:    0x00000000
<bochs:11> xp 0x7c00
[bochs]:
0x00007c00 <bogus+       0>:    0x00000000
<bochs:12>
```

▲图 3-19　bochs 中 xp 指定显示单元

图 3-19 中尝试了用 b 和 w 来显示内存，大家可以看出，指定了显示单元后，以后的输出就以此显示单元为准，不会自动恢复为默认的 4 字节。

此处的 0x7c00 是空值 0，这是因为 MBR 还没有被加载到此地址，MBR 是由 BIOS 来加载的，BIOS 目前还没有运行呢。

为了让大家更好地理解 BIOS 是怎样被执行的，也就是计算机中第一个软件是怎样开始的，咱们还是先看图 3-17。在图的上面第 5 行，显示的是下一条待执行的指令，这是程序计数器（PC）中的值，在 x86 上的程序计数器是指 cs：ip。大家看，cs 是 0xf000,ip 是 fff0，所以最终地址是 0xffff0，这是 BIOS 的入口地址，即低端 1M 内存最顶端的 16 字节。忘记的同学往前翻看表 1-1 实模式下的内存布局。在右边的 jmp far f000:e05b，是指此内存 0xffff0 处的内容，在内存中，内容不是普通数据，就是指令，在此处就是条跳转指令。这是咱们之前分析 BIOS 时所说的，1M 内存的最顶端只有 16 字节，肯定容不下完整的 BIOS 代码，必然是条跳转指令。由此可见，果然没错，它是跳转到 0xf000:e05b 的地方了。咱们需要验证一下内存 0xffff0 处的内容是不是 jmp far f000:e05b。

验证的方法有多种，不过先来个简单粗暴可依赖的，咱们查看此处内存的内容是什么。内容如图 3-20 所示。

```
<bochs:2> xp/2 0xffff0
[bochs]:
0x000ffff0 <bogus+       0>:    0x00e05bea      0x2f3131f0
<bochs:3>
```

▲图 3-20　通往 BIOS 的 jmp 跳转

由于默认 xp 以 4 字节来显示，所以 xp 中斜杠后面指定的数字 2，最终会让 xp 显示 8 个字节。提醒一下，咱们 bochs 模拟的是 x86 平台，它是小端字节序。咱们只看 1 个 4 字节，先从低地址看，最低位是 ea，这是直接绝对远转移 jmp far 的机器码，高位的是 0xe05b，这是 jmp far 的操作数，待跳转到的地址，如果忘记指令格式的同学，赶紧到前面找到 IA32 指令格式回忆一下。

这与程序计数器（cs:ip）中指定的内容是吻合的，不过咱们还是不太放心，也许您说，万一 0x00e05bea 只是普通数据呢，我又对机器码不熟，不许忽悠我。为了打消您的疑虑，那再用一种方法来验证一下吧。

在“Debugger control”类中，有个命令 U，它用来将内存数据反汇编成指令。咱们看一下此命令的帮助。help u 回车后效果如图 3-21 所示。

```
<bochs:12> help u
u|disasm [/count] <start> <end> - disassemble instructions for given linear addr
ess
    Optional 'count' is the number of disassembled instructions
u|disasm switch-mode - switch between Intel and AT&T disassembler syntax
u|disasm hex on/off - control disasm offsets and displacements format
u|disasm size = n - tell debugger what segment size [16|32|64] to use
       when "disassemble" command is used.
<bochs:13>
```

▲图 3-21　bochs 的反汇编指令 u

大概意思是说，u 和 disasm 是一样的命令，用哪个都行，其用法是在后面跟需要汇编的指令数、起始线性地址、终止线性地址。由于我们在实模式下，线性地址就是物理地址。键入 u/1 0xffff0 回车，效果如图 3-22 所示。

```
<bochs:1> u/l 0xffff0
000ffff0: (                    ): jmp far f000:e05b         ; ea5be000f0
<bochs:2>
```

▲图 3-22　反汇编效果

果然没有忽悠大家，地址 0xffff0 处的内容确实是指令，并且是 jmp far f000：e05b。这下大家对理解 BIOS 启动应该更深刻一些了。

这个小例子完成了，大家如果对此过程有任何疑惑，希望在下面的内容中找到答案。

在 bochs 提供的命令中，大部分调试命令都有简写和缩写，它们用或运算字符'|'彼此分隔，所以它们都是同一个功能，只是不同的别名，用哪一个都可以。下面按照上面提到的几大类，对常用的调试指令做个大概的介绍。

- “Debugger control”类

q|quit|exit，左边三个命令任意一个都能退出调试状态，关闭虚拟机，一般用 q 最简单。

set 是指令族，咱们通常用 set 设置寄存器的值，这个较常用。

（1）例如 set reg = val。可以设置的寄存器包括通用寄存器和段寄存器。

（2）也可以设置每次停止执行时，是否反汇编指令：set u on|off。

show 是指令族，有很多子功能，咱们常用的就下面这 3 个。

1．show mode

每次 CPU 变换模式时就提示，模式是指保护模式、实模式，比如从实模式进入到保护模式时会有提示。

2．show int

每次有中断时就提示，同时显示三种中断类型，这三种中断类型包括“softint”、“extint”和“iret”。

可以单独显示某类中断，如执行 show softint 只显示软件主动触发的中断，show extint 则只显示来自外部设备的中断，show iret 只显示 iretd 指令有关的信息。

3．show call

每次有函数调用发生时就会提示。

traceon|off 如果此项设为 on，每次执行一条指令，bochs 都会将反汇编的代码打印到控制台，这样在单步调试时免得看源码了。

```
u|disasm [/num] [start] [end]
```

将物理地址 start 到 end 之间的代码反汇编，如果不指定地址，则反汇编 EIP 指向的内存。num 指定反汇编的指令数。

setsize = 16|32|64 在使用反汇编命令时，用来告诉调试器段的大小。

set 指令也会设置在停止时是否反汇编命令。前面 set 命令中有说过。

ctrl+c 中断执行，回到 bochs 控制台。

- “Execution control”类

c|cont|continue，左边列出的三个命令都意为向下持续执行，若没断点则一直运行下去。最常用的是 c。

s|step [count] 执行 count 条指令，count 是指定单步执行的指令数，若不指定，count 默认为 1。此指令若遇到函数调用，则会进入函数中去执行。最常用的是 s。

p|n|next 执行 1 条指令，若待执行的指令是函数调用，不管函数内有多少指令，把整个函数当作一个整体来执行。最常用的是 n。

- “Breakpoint management”类

以地址打断点：

vb|vbreak [seg：off] 以虚拟地址添加断点，程序执行到此虚拟地址时停下来，注意虚拟地址是“段：段内偏移”的形式。最常用的是 vb。

lb|lbreak [addr]以线性地址添加断点，程序执行到此线性地址时停下来。最常用的是 lb。

pb|pbreak|b|break [addr]以物理地址添加断点。程序执行到此物理地址时停下来。b 比较常用。

以指令数打断点：

sb [delta] delta 表示增量，意味再执行 delta 条指令程序就中断。

sba [time] CPU 从运行开始，执行第 time 条指令时中断，从 0 开始的指令数。

以读写 IO 打断点：

watch 也有子命令，常用的是这两个。

watch r|read [phy_addr] 设置读断点，如果物理地址 phy_addr 有读操作则停止运行。

watch w|write [phy_addr] 设置写断点，如果物理地址 phy_addr 有写操作则停止运行。此命令非常有用，如果某块内存不知何时被改写了，可以设置此中断。

watch 显示所有读写断点。

unwatch 清除所有断点。

unwatch [phy_addr] 清除在此地址上的读写断点。

blist 显示所有断点信息，功能等同于 info b。

bpd|bpe [n] 禁用断点（break point disable）/启用断点（break point enable），n 是断点号，可以用 blist 命令先检查出来。

d|del|delete [n] 删除某断点，n 是断点号，可以用 blist 命令先检查出来。D 最常用。

- “CPU and memory contents”类

x /nuf [line_addr] 显示线性地址的内容。n、u、f 是三个参数，都是可选的，如果没有指定，则 n 为 1，u 是 4 字节，f 是十六进制。解释如下。

n 显示的单元数

u 每个显示单元的大小，u 可以是下列之一：

（1）b 1 字节；

（2）h 2 字节；

（3）w 4 字节；

（4）g 8 字节。

f 显示格式，f 可以是下列之一：

（1）x 按照十六进制显示；

（2）d 十进制显示；

（3）u 按照无符号十进制显示；

（4）o 按照八进制显示；

（5）t 按照二进制显示；

（6）c 按照字符显示；

（7）s 按照 ASCIIz 显示；

（8）i 按照 instr 显示。

xp /nuf [phy_addr] 显示物理地址 phy_addr 处的内容，注意和 x 的区别，x 是线性地址。

setpmem [phy_addr] [size] [val] 设置以物理内存 phy_addr 为起始，连续 size 个字节的内容为 val。此命令非常有用，在某些情况下不易调试时，可以在程序中通过某个地址的值来判断分支，需要用 setpmem 来配合。

注意啦，size 最多只能设置 4 个字节宽度的数据，如果 size 大于 4 便会报错：Error：setpmem：bad length value = 8。size 小于等于 4 是正确的，setpmem 0x7c00 4 0x9。

r|reg|regs|registers 任意四个命令之一便可以显示 8 个通用寄存器的值+eflags 寄存器+eip 寄存器。r 是我常用的查看寄存器的命令。

ptime 显示 Bochs 自启动之后，总执行指令数。其实这个命令不常用，感兴趣的同学可以用 ptime 和“sb 指令数”来验证结果是否正确。

print-stack [num] 显示堆栈，num 默认为 16，表示打印的栈条目数。输出的栈内容是栈顶在上，低地址在上，高地址在下。这和栈的实际扩展方向相反，这一点请注意。

?|calc 内置的计算器。

info 是个指令族，执行 help info 时可查看其所有支持的子命令，如下：

info pb|pbreak|b|break 查看断点信息，等同于 blist。

info CPU 显示 CPU 所有寄存器的值，包括不可见寄存器。

info fpu 显示 FPU 状态。

info idt 显示中断向量表 IDT。

info gdt [num]显示全局描述符表 GDT，如果加了 num，只显示 gdt 中第 num 项描述符。

info ldt 显示局部描述符表 LDT。

info tss 显示任务状态段 TSS。

info ivt [num]显示中断向量表 IVT。和 gdt 一样，如果指定了 num，则只会显示第 num 项的中断向量。如果各位大侠想知道 BIOS 在中断向量表中建立了哪些中断，执行此命令就可以看到了，还有相关说明呢。图 3-23 所示是我的部分截屏。

```
<bochs:14> info ivt
INT# 00 > F000:FF53 (0x000fff53) DIVIDE ERROR ; dummy iret
INT# 01 > F000:FF53 (0x000fff53) SINGLE STEP ; dummy iret
INT# 02 > F000:FF53 (0x000fff53) NON-MASKABLE INTERRUPT ; dummy iret
INT# 03 > F000:FF53 (0x000fff53) BREAKPOINT ; dummy iret
INT# 04 > F000:FF53 (0x000fff53) INT0 DETECTED OVERFLOW ; dummy iret
INT# 05 > F000:FF53 (0x000fff53) BOUND RANGE EXCEED ; dummy iret
INT# 06 > F000:FF53 (0x000fff53) INVALID OPCODE ; dummy iret
INT# 07 > F000:FF53 (0x000fff53) PROCESSOR EXTENSION NOT AVAILABLE ; dummy iret
INT# 08 > F000:FEA5 (0x000ffea5) IRQ0 - SYSTEM TIMER
INT# 09 > F000:E987 (0x000fe987) IRQ1 - KEYBOARD DATA READY
INT# 0a > F000:E9DF (0x000fe9df) IRQ2 - LPT2
INT# 0b > F000:E9DF (0x000fe9df) IRQ3 - COM2
INT# 0c > F000:E9DF (0x000fe9df) IRQ4 - COM1
INT# 0d > F000:E9DF (0x000fe9df) IRQ5 - FIXED DISK
INT# 0e > F000:EF57 (0x000fef57) IRQ6 - DISKETTE CONTROLLER
INT# 0f > F000:E9DF (0x000fe9df) IRQ7 - PARALLEL PRINTER
INT# 10 > C000:014A (0x000c014a) VIDEO
INT# 11 > F000:F84D (0x000ff84d) GET EQUIPMENT LIST
INT# 12 > F000:F841 (0x000ff841) GET MEMORY SIZE
INT# 13 > F000:E3FE (0x000fe3fe) DISK
INT# 14 > F000:E739 (0x000fe739) SERIAL
INT# 15 > F000:F859 (0x000ff859) SYSTEM
INT# 16 > F000:E82E (0x000fe82e) KEYBOARD
```

▲图 3-23　中断向量表 ivt 描述

info flags|eflags 显示状态寄存器，其实在用 r 命令显示寄存器值时也会输出 eflags 的状态，还会输出通用寄存器的值，我通常会用 r 来看。

sreg 显示所有段寄存器的值。

dreg 显示所有调试寄存器的值。

creg 显示所有控制寄存器的值。

info tab 显示页表中线性地址到物理地址的映射。

page line_addr 显示线性地址到物理地址间的映射。

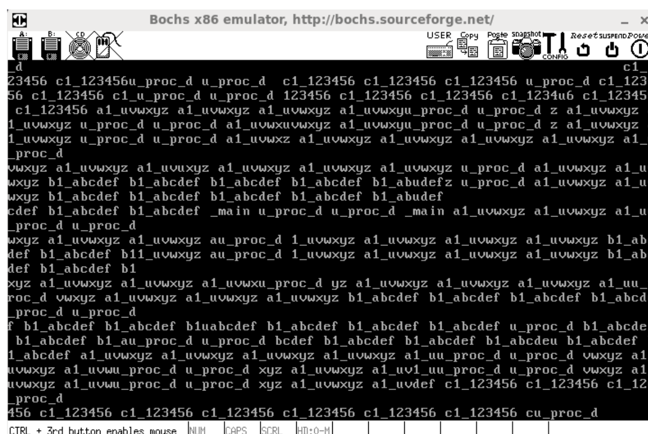
好啦，以上的介绍说长不长，说短不短，希望对大家有所帮助。大家试着在 bochs 中练习一下，一会咱们就要实战一把，看看如何利用这些调试利器解决实际问题。

3.4.2　bochs 调试实例

下面这个例子是我在实际开发过程中遇到 bug 时的调试思路，如果您的开发经验比较丰富可以忽略本节，本节主要针对非专业开发同学而写。

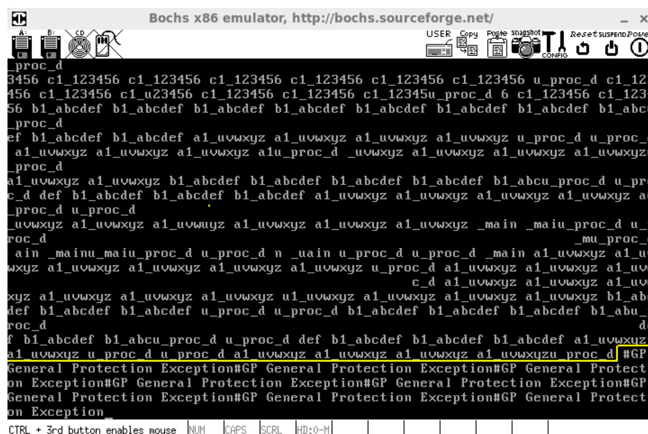
下面把这个调试过程和大家分享，希望对于开发经验不足的读者能起到抛砖引玉的作用。

背景提要：这两张图是多个线程和一个进程在并发执行时的情况，图 3-24 所示是运行之初正常的情况，图 3-25 所示是抛异常的结果，大家就不要在这两张图片中玩“找不同”了，图 3-25 中画线部分以下的是开始抛异常的部分。



▲图 3-24　多个线程与进程并发执行

大部分的报错是逻辑错误，如果程序在运行开始时就报错还比较好办，若像图 3-24 那样运行之初还是好好的，执行一定次数后，运行大概 1 秒才报出如图 3-25 所示的一般保护性异常，这种运行中的错误似乎让人有些无从查起，尤其是在没有宿主系统的情况下。



▲图 3-25　突然出现 GP 错误

进入正题，下面是此报错的调试过程，注意，这个例子是保护模式下的调试，现在虽然还没讲保护模式，但现在介绍的是调试思路，大家只要有个宏观认识就可以了。

首先打开 show int，此命令是让 bochs 在中断发生时输出提示，因为上面的 GP 报错是某种原因引发了中断而进入中断处理程序时打印的，如果您知道是某种中断导致的问题，可以直接用 show extint、show softint 或 show iret 针对某一类中断排查。不过还是直接用 show int 这种一网打尽的方式比较省心。中断的概念咱们会在后面讲，这里先知道这个情况就行了：屏幕上能够输出“#GP General Protection Exception”，是因为发生了异常，我们需要知道发生异常之前程序的状态。

好啦，下面开始在 bochs 中调试。

```
<bochs:1> show int
show interrupts tracing (extint/softint/iret): ON
show mask is: softint extint iret
<bochs:2> c
.............
..........中间内容略
............
...............
00020182862: iret 002b:c00036cc (0xc00036cc)
00020182900: softint 0008:c00021e7 (0xc00021e7)
00020183349: iret 002b:c00036cc (0xc00036cc)
00020183350: exception (not softint) 0008:c0002029 (0xc0002029)
00020183503: iret 0008:c000223c (0xc000223c)
```

```
00020183518: exception (not softint) 0008:c0001e1b (0xc0001e1b)
^CNext at t=38352358
(0) [0x000000002246] 0008:c0002246 (unk. ctxt): mov dx, 0x03d5         ; 66bad503
<bochs:3>

******************以下内容为解释说明，非程序输出*************
```

bochs 中的输出会停在此处，mov dx， 0x03d5 是下一条待执行的指令不必管它，要关注上面已执行过的输出中的最后一句：

```
00020183518: exception (not softint) 0008:c0001e1b (0xc0001e1b)
```

看上去是执行到 0008：c0001e1b 处发生的中断。在它行首的冒号左边一连串数字 00020183518 表示自启动后执行了指令的数目（ptime 指令也是输出自启动后执行的指令数）。

这就是关键的部分了，这是程序运行过程中最后一个中断，初步判断它就是引发 CPU 抛 GP 异常的中断。

那在此中断之前发生了什么？哪个语句引起了此中断呢？一个可行的想法是可以在调试指令 sba 中设置一个时间点断点，而设置的值便为 00020183518 的上一条指令数。

不过先不急，注意看，这个中断所调用的中断处理函数，其选择子是 08，段内地址是 0xc0001e1b。因为这是在保护模式下，所以段基址寄存器中的内容不再是地址，而是个选择子。不过在此大家就先理解为段基址就行了，这不影响大家理解本节后面的内容。我还想看看这个地址是哪个函数，再发挥一下 nm 命令的威力。

nm 命令用来列出可执行文件的符号表及其地址，在不加参数的情况下，默认输出“地址”“符号类型”“函数名”。用法：nm 二进制可执行文件。注意，纯二进制文件不支持，支持 elf 文件，纯二进制文件中不包含符号表。

grep 命令是在输入信息中匹配出含有“参数”的行并打印出来。用法：grep “参数字符串”输入文件或通过管道“|”。

由于 nm 会输出文件内所有符号信息，所以用 grep 将此地址过滤出来。

nm build/kernel.bin |grep 1e1b 回车

以下是输出。

c0001e1b t intr##0x0d##entry

ok，上面第一列的 c0001e1b 是地址，t 代表这是位于代码段的符号，intr##0x0d##entry 是符号名。这是我自定义的函数名，规则是 intr##中断号##entry，以后会介绍，现在只要关注中间的中断号就行啦。符号名中的 0xd，代表 13 号中断处理函数，果然是 GP 错误，即"#GP General Protection Exception"。所以证明这最后一个中断确实是在报 GP 错误，我们可以放心地继续调试啦。

下面这个调试步骤是多此一举的，就是为使用 lb 指令验证下指令数是否正确。不过当代码量大到一定程度时还是要小心谨慎，以一步一回头的方式来查看比较好。

先在 0xc0001e1b 处设置个断点。

```
****************  以上说明结束，下面是 bochs 控制台输出************
<bochs:1> lb 0xc0001e1b
<bochs:2> c
(0) Breakpoint 1, 0xc0001e1b in ?? ()
Next at t=20183518
(0) [0x000000001e1b] 0008:c0001e1b (unk. ctxt): nop                    ; 90
<bochs:3>
******************以下内容为解释说明，非程序输出************
```

看上面的 Next at t=20183518，和上一步调试中的冒号左边的数值是一致的。

也就是说，中断是在执行第 20183518 条指令执行的，那么发生异常从而引发中断的指令一定是在上一条，即（20183518-1）处，那我们可放心地用 20183517 作为时间点断点的值了，下面继续。

此时屏幕的输出可以参看图 3-23，内容差不多，还没有开始打印"GP General Protection Exception"，到这为止一切貌似正常。下面用 sba 指令设置时间点断点。

```
****************  以上说明结束，下面是 bochs 控制台输出************
```

```
<bochs:1> sba 20183517
Time breakpoint inserted. Delta = 20183517
<bochs:2> c                    ******继续执行
(0) Caught time breakpoint     ******  提示捕捉到了断点
Next at t=20183517

******************以下内容为解释说明，非程序输出************
```

Next at t=20183517，下一个要执行的指令是第 20183517 条。也就是下面马上要执行的指令是引发中断的祸根。

```
**************  以上说明结束，下面是 bochs 控制台输出************

(0) [0x00000000225d] 0008:c000225d (unk. ctxt): mov byte ptr gs:[bx], cl  ; 6567880f

******************以下内容为解释说明，非程序输出************
```

注意看这条 mov 指令，“mov byte ptr gs：[bx]，cl”，bx 作为偏移地址。不急，先查看下 ebx 寄存器的值。

```
**************  以上说明结束，下面是 bochs 控制台输出************

<bochs:3> r
eax: 0xc010caca -1072641334
ecx: 0x00000031 49
edx: 0xc00003d5 -1073740843
ebx: 0xc0009594 -1073703532            bx 为 0x9594
esp: 0xc0103de4 -1072677404
ebp: 0xc0103e40 -1072677312
esi: 0x00000000 0
edi: 0x00000000 0
eip: 0xc000225d
eflags 0x00000283: id vip vif ac vm rf nt IOPL=0 of df IF tf SF zf af pf CF

******************以下内容为解释说明，非程序输出************
```

再查看下 gs 的值，gs 值为 0x0018。

```
**************  以上说明结束,下面是 bochs 控制台输出************

<bochs:4> sreg
es:0x0010, dh=0x00cf9300, dl=0x0000ffff, valid=1
Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed
cs:0x0008, dh=0x00cf9900, dl=0x0000ffff, valid=1
Code segment, base=0x00000000, limit=0xffffffff, Execute-Only, Non-Conforming, Accessed, 32-bit
ss:0x0010, dh=0x00cf9300, dl=0x0000ffff, valid=7
Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed
ds:0x0010, dh=0x00cf9300, dl=0x0000ffff, valid=1
Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed
fs:0x0000, dh=0x00001000, dl=0x00000000, valid=0
gs:0x0018, dh=0x00c0930b, dl=0x80000008, valid=1
Data segment, base=0x000b8000, limit=0x00008fff, Read/Write, Accessed
ldtr:0x0000, dh=0x00008200, dl=0x0000ffff, valid=1
tr:0x0020, dh=0xc0808b00, dl=0x45a0006b, valid=1
gdtr:base=0x00000900, limit=0x37
idtr:base=0x00004180, limit=0x407

******************以下内容为解释说明，非程序输出************
```

最简单排错的方法是用 x 命令查看该处的内存怎么啦。

```
**************  以上说明结束，下面是 bochs 控制台输出************

<bochs:5> x gs:ax
WARNING: Offset 0000CACA is out of selector 0018 limit (00000000...00008fff)!

******************以下内容为解释说明，非程序输出************
```

上面果然报错啦，提示超出了段界限。

gs 所指向的显存段的 limit 是多少呢？马上查看下该段的段描述符。

根据 gs 的值 0x18，由于低 3 位是 rpl 和 TI 位，所以可知选择索引值，其二进制是 11，即十进制 3。

直接查看 gdt 中索引为 3 的段描述符。

```
**************  以上说明结束，下面是 bochs 控制台输出************

<bochs:6> info gdt 3
Global Descriptor Table (base=0x00000900, limit=55):
GDT[0x03]=Data segment, base=0x000b8000, limit=0x00008fff, Read/Write, Accessed

******************以下内容为解释说明，非程序输出************
```

果然 limit 为 0x8fff，寄存器 bx 的值为 0x9594，用 bx 做偏移量导致越界，所以引发 GP 异常。找到错误后，为保险起见再执行一下是否会引发异常。按下 s 键执行上面的指令“mov byte ptr gs:[bx], cl”。

```
**************  以上说明结束，下面是 bochs 控制台输出************

<bochs：7> s
Next at t=20183518

******************以下内容为解释说明，非程序输出************
```

果然下面的是中断处理程序的代码地址 0008：c0001e1b。

```
**************  以上说明结束，下面是 bochs 控制台输出************

(0)[0x000000001e1b] 0008:c0001e1b (unk. ctxt):
    nop                   ; 90

<bochs:8> s
Next at t=20183519
(0)[0x000000001e1c] 0008:c0001e1c (unk. ctxt): push
    ds                ; 1e
<bochs:9>
```

问题定位后，接下来就要检查下 bx 的值为什么会超过段界限了，因为我们操作的是 gs 寄存器，此寄存器在我们项目中用来存储显存段选择子，所以八成是字符串输出函数中对 bx 没有限制导致的。而我们的函数是 C 写的，从汇编到 C 去排查，很多时候并不需要了解编译规则，只要定位了方向，在 C 代码中通常盯上一会便能够让 bug 显现出来啦。好啦，本节到此结束，同学们收摊儿啦。

3.5 硬盘介绍

要了解硬盘，也要了解其历史。硬盘属于存储介质，要说这存储介质的发展史，还得从甲骨文、山洞壁画开始说起呢。这是我的老师当初讲硬盘时说的，不过咱们没那么多精力，直接关注硬盘简史就好啦。

3.5.1　硬盘发展简史

个人觉得，在计算机世界中，实现随机存取具有划时代的意义，程序中的算法不用再把存储时间考虑进去，访问任意数据所用的时间几乎相等，这一改之前的存储设备其存取时间呈线性的历史。而改变这一历史的时间是 1956 年 9 月，世界上诞生了第一台磁盘存储设备 IBM 350 RAMAC（Random Access Method of Accounting and Control），没错，又是 IBM，这就是蓝色巨人的魅力，在几十年前已经为科技领航。此设备用磁头来读写数据，用盘片来存储数据，以后的硬盘都是以这样的模式发展。这个磁头可以直接移动到盘片上的任何一块存储单元，从而实现了随机存储。虽然它的总容量只有 5MB，但是限于当时的制造工艺水平，一共使用了 50 个直径为 24 英寸的盘片，摞起来的体积相当于冰箱那么大。在这些盘片表面都涂有用于存储数据的磁性物质，它们摞起来被固定在一起，在电机的带动下，绕着同一个轴旋转。今天看起来它显得过于笨重且容量小得以至于没什么用，但当时可是高大上的玩意儿，所以它在那时主要用于高精尖领域呢，如航空业、银行业、医学领域及太空领域。

1968 年由 IBM 首次提出了“温彻斯特/Winchester”的技术，这可是个了不起的技术。“温彻斯特”技术的精髓是：“镀磁盘片在密封空间中高速自转，磁头悬浮在盘片上方，固定在磁头臂上沿盘片径向移动”。

磁头不与盘片接触也不应该接触，这是最容易想象的，如果磁头与盘片接触，在高速转动下摩擦，什

么材料都会磨损，数据自然就丢啦。另外，盘片自转速度是存取数据速度的关键，如果磁头与盘片接触，受摩擦力的影响，想快也快不了。一个可行的方案是让磁头在盘片上方“悬浮”，与盘片保持非常近的距离，类似咱们物理实验中的气垫导轨。盘片高速旋转会产生气流，磁头在这种气流下像飞碟一样悬浮，这样就保证了不会与盘片有摩擦。磁头被固定在磁头臂上，它能沿盘片径向移动，由于磁头和盘片各自的运动，再加上如此近的距离，所以哪怕一点灰尘都会造成磁盘的损伤。于是，磁头、盘片被密封在了一个盒子里。今天的硬盘依然是这样的结构，如图 3-26 所示。

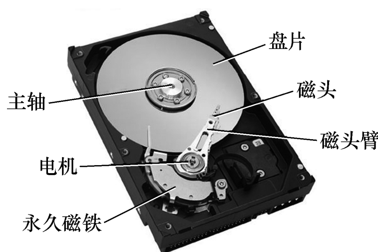
但这只是提出了这样的构想，此时还没有制造出这样的硬盘呢。思想是人家提出的，别人还真玩不转，所以制造还得由人家 IBM 亲自来完成。世界上第一块基于 “温彻斯特”技术的硬盘，在 1973 年诞生，它就是所有硬盘的源头 IBM 3340，容量 60MB，由两个 30MB 的存储单元拼合而成。从此硬盘技术的发展便有了成形的结构基础。所以，今天的硬盘也称为温盘，在一般的可编程中断控制器上连接硬盘的引脚都标有温彻斯特硬盘呢。

“温彻斯特”这个名字还有个典故，一些重大发明，其名字背后大多数都有个小故事。由于 IBM 3340 拥有两个 30MB 的存储单元，而当时一种很有名的“温彻斯特来复枪”的口径和装药也恰好包含了两个数字“30”。于是这种硬盘的内部代号就被定为“温彻斯特”。据说电影《终结者》中施瓦辛格拿的枪便是“温彻斯特来复枪”。

硬盘发展到今天，这几十年来都是靠容量不断提升才打败竞争对手存活下来的。可是速度一直是其最大的敌人，家用电脑 10 年前硬盘主流转速是 7200 转/分钟，今天依然如此。硬盘的随机存取是靠磁头臂不断移动实现的，磁头臂移动到目标位置的时间称为寻道时间，如果存储的数据不连续，这一块那一片的，磁头就得不断调整位置，这是机械式硬盘不可避免的，这便是硬盘的瓶颈所在，所以一般的硬盘都将寻道时间作为重要参数。

在众多竞争对象当中，也有一款顽强地活了下来，它就是 SSD 固态硬盘，人家也有几十年的历史了。

SSD 固态硬盘为我们带来了全新的解决方案，如图 3-27 所示。



▲图 3-26　硬盘内部结构


▲图 3-27　SSD 固态硬盘

怎么样，看起来像“大块的”内存条，看不到机械部件。传统硬盘是机械式硬盘，而 SSD 同 U 盘一样，基于闪存，也就是 Flash Memory，它不存在机械部件，这就是它快的原因。SSD 全称是 Solid State Disk，翻译过来就是固态硬盘。不过这个“固态”和“三态”无关，不是说其他硬盘是“液态”或者是“气态”的，而是突显了其优势：稳定与牢固。

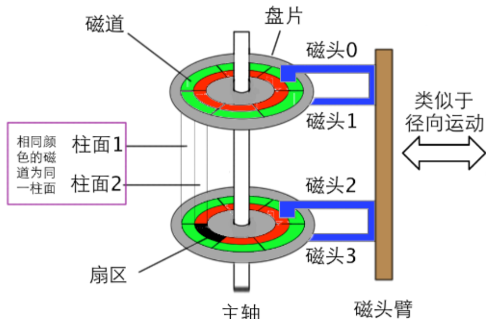
它能存活下来是有其道理的，存在即合理。尽管温彻斯特硬盘为了提速，内部加了很多缓存，应用了各种优化寻道的方法，甚至接口已经变成了串口，但主流磁盘转速还是 7200 转，速度却提升有限。于是固态硬盘的优势开始显山露水，其最大的优点是速度快，但缺点也明显，容量低，价格也很贵。所以在个人电脑中很少见到 ssd 硬盘，目前主要还是运用在要求存储速度较快的生产环境中，如数据库系统、分布式管理中心 zookeeper。

3.5.2　硬盘工作原理

为了讲清楚硬盘的工作原理，我用画图板花了 1 个小时画了这张示意图，希望对大家有所帮助。大家请参看图 3-28。

左边的主轴上有两个盘片，其实不止两个，这里示意性地只画了两个。盘片固定在主轴上随主轴高速

转动，目前主流个人电脑硬盘上的转速是 7200 转/分钟。每个盘片分上下两面，每面都存储数据，每个盘面都各由一个磁头来读取数据，故一个盘片上对应 2 个磁头。由于盘面与磁头是一一对应的关系，故用磁头号来表示盘面。磁头编号从上到下以 0 开始计数，所以用磁头 0 代表第一个盘面。磁头不会自己在盘片上移动，它需要被固定在右边的磁头臂上，在磁头臂的带动下，沿着盘片边缘向圆心的方向来回摆动，注意，摆动的轨迹是个弧，并不是绝对径向地直来直去。一方面这是因为磁头臂是由步进电机驱动的，磁头臂一端是步进电机主轴，另一端是磁头。步进电机每次都会转动一个角度，所以带动磁头臂在“画圆”，而磁头位于磁头臂的另一端，所以也跟着呈钟摆运动，运动轨迹是个弧线，并不是直线。所以，图 3-28 中磁头臂中标注了“类似于”径向运动，这就是“类似”的原因。另一方面，磁头读取数据也不需要做直来直去的移动，能否找到数据，只跟它最终落点有关，和中间路径形状是没关系的。所以，一方面盘片的自转，另一方面磁头的摆动，这两种动作的合成，使磁头能够读取盘片任意位置的数据。



▲图 3-28　机械式硬盘示意图

说完了硬盘内部各部件的运动，再说下存储逻辑。盘片表面是用于存储数据的磁性介质，为了更有效管理磁盘，这些磁性介质也被“格式化”成易于管理的格局，即将整个盘面划分为多个同心环，以圆心画扇形，扇形与每个同心环相交的弧状区域作为最基本的数据存储单元。这个同心环就称为磁道，而同心环上的弧状区域是扇形的一部分，故称之为扇区，它作为我们向硬盘存储数据的最基本单位，大小是 512 字节。我们写入的数据最终是写进了磁道上的扇区中。注意啦，我上面说的磁道是个环，不是线，很多教科书上介绍磁道时都简单画了个圆圈，这容易让人误解磁道是条线，线上可无法存储数据，“环”是有横截面的，数据就存储这些“面积”中。磁头臂带动磁头在盘片上方移动，就是在找磁道的位置，盘片高速自转，就是在磁道内定位扇区。大家看看图 3-28，配合着想像力理解一下。

磁道的编号和磁头一样也是从 0 开始的。相同编号的磁道组成的管状区域就称为柱面。图 3-28 中，两个盘片上编号相同的磁道，它们之间用灰色直线连接起来的部分，很像柱子的弧形表面，所以称柱面。如果盘片非常多的话，“柱面”就显得非常形象了。

柱面这个看似抽象的概念有什么用呢？前面介绍过了，机械式硬盘的寻道时间是整个硬盘的瓶颈，为了减少寻道时间，就尽量在存储上下工夫。寻道，简而言之就是磁头在磁道间跳转，跳转所需要的时间称为寻道时间。如果待写入的数据小于一个磁道的剩余容量，将来再读出来的时候，磁头只定位到该磁道就行了。这时候的寻道只是一次。如果待写入的数据要占用多个磁道时，除了写的时候要变换磁头到新磁道，将来读出来的时候也需要变换磁道，也就是需要多次寻道才能完成数据的完整读写。既然寻道对机械式硬盘速度影响较大，那原则上就尽量减少寻道次数。于是按柱面存取的想法就诞生了：柱面中的磁道是相同编号，编号相同则意味着磁道在盘面上的位置相同，要定位到同一柱面中的磁道，所有磁头位置都一样，于是磁头不用再移动了。是不是很赞呢？

按照这种想法写数据：当 0 面上的某磁道空间不足时，其他数据写入第 1 面相同编号的磁道上。若新磁道空间还是不足，再写第 2 面相同编号的磁道上，直到同一柱面上的磁道（所有盘面上的编号相同的磁道）都不够用时才会写到新的柱面上。所以，在这一点上，盘面越多，硬盘越快。

扇区编号与盘面和磁道不同，各磁道内的扇区是以 1 为起始编号的，并且只限于本磁道内有效，所以各个磁道间的扇区编号都相同，下限都是以 0 起，上限就不一定了，取决于各厂商的工艺，不过大多数情况下一个磁道中有 63 个扇区。磁头如何找到所需要的扇区呢？即使是按照编号查找也得有个地方让磁头去比对编号吧。原来，扇区是有自己的“头部”，头部之后的部分才是存储数据的 512 字节。头部中包含了扇区自身的信息，哪些信息能唯一定位一个扇区呢？答案是磁头号、磁道号和扇区号。

接下来咱们看看，目前咱们的主板是怎样支持硬盘的。

之前和大家讨论显卡时说过了，CPU 小朋友很低调，不擅长和陌生人交流，只和熟悉的 IO 接口通信，

由 IO 接口代 CPU 向下传达旨意。CPU 是怎样和硬盘打交道的呢?换句话说，针对硬盘的 IO 接口是什么？答案就是硬盘控制器。

硬盘控制器同硬盘的关系，如同显卡和显示器一样，它们都是专门驱动外部设备的模块电路，CPU 只同它们说话，由它们将 CPU 的话转译给外部设备。这是它们的共同点，但不同的是显卡和显示器是分开的，硬盘控制器和硬盘是连接在一起的，至少在我接触计算机以来一直就是这样的，从未变过，也不曾怀疑过。但我有天被告知，在很久很久以前，硬盘控制器和硬盘也是分开的，就像显卡和显示器一样。显示器太大了，怎么能装在机箱里呢，当然要和显卡分开了。可硬盘控制器和硬盘都那么小，为什么还要分开呢？

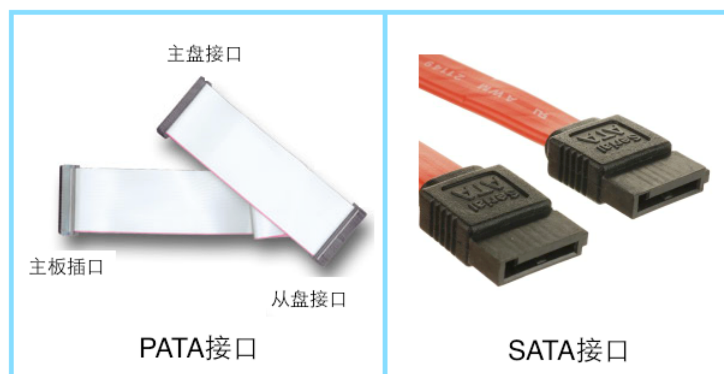
是啊，为什么呢，我也不知道，可它们在很久以前就是分开的。后来业界的几个老大合作开发出一种新的接口，这样才将硬盘和硬盘控制器整合在一起，为了突显“整合”之意，硬盘控制器和硬盘终于在一起了，这种接口便称为集成设备电路（Integrated Drive Electronics，IDE）。随着 IDE 接口标准的影响力越来越广泛，全球标准化协议将此接口使用的技术规范归纳成为全球硬盘标准，这样就产生了 ATA（Advanced Technology Attachment）。不过由于 IDE 这个名字已经叫开了，所以大家依然习惯称硬盘为 IDE 硬盘。计算机发展非常快，新老交替的现象层出不穷，以至于后辈的出现常常把前辈的名字都给改了。这不，前几年刚出道的硬盘串行接口（Serial ATA，SATA），由于其是串行，所以之前的 ATA 接口只好称为并行 ATA，即（Parallel ATA，PATA）。

以前一般的主机只支持 4 个并口硬盘，但自从出现串口硬盘后，情况就变了，支持多少块硬盘，取决于主板的能力。有的主板同时兼容这两种接口，如图 3-29 所示。



▲图 3-29　硬盘的并口与串口接口

图中 6 个并排的小接口都是 SATA，插槽里面呈字符 L 形，这就是它的标志。左边的长方形，里面有好多针的接口是 PATA 接口，也就是之前传统的硬盘是插在此接口上面的。下面把两个接口对应的线缆贴出来对比一下，如图 3-30 所示。



▲图 3-30　硬盘的并行接口与串行接口

这两种线缆完全不同，左边 PATA 接口的线缆也称为 IDE 线，一个 IDE 线上可以挂两块硬盘，一个是主盘（Master），一个是从盘（Slave）。标有“主板插口”的那一头要插在图 3-29 中 PATA 接口处。在之前，主盘从盘的分工很明显，很多工作都要以主盘为主，比如系统就要装在主盘上。到后来系统兼容性越来越好，以至区别不明显了。前面说过了，一个主板支持这样的 4 块 IDE（PATA）硬盘，所以主板上提供两个 IDE 插槽。这两个接口也是以 0 为起始编号的，一个称为 IDE0，另一个称为 IDE1。不过按 ATA 的说法，这两个插槽称为通道，IDE0 叫作 Primary 通道，IDE1 叫作 Secondary 通道。即使主板上安装的是 SATA 硬盘，它也兼容 PATA 的编程接口，向上兼容是计算机能源源不断向前发展的根基。所以，后面给出的端口号也将按照 PATA 这两个通道来分组给出。

多说一句，这里所说的主盘 master、从盘 slave 别和 Primary 通道、Secondary 通道搞混了，通道是 channel，不是 disk，每个通道上分别有主盘和从盘。

3.5.3　硬盘控制器端口

差不多该说如何控制硬盘了。硬盘控制器属于 IO 接口，前面在讲解显卡之前有介绍过相关知识，不再赘述。

让硬盘工作，我们需要通过读写硬盘控制器的端口，端口的概念在此重复一下，端口就是位于 IO 控制器上的寄存器，此处的端口是指硬盘控制器上的寄存器。

下面列出了部分端口，对于我们今后的应用，这几个端口足够了。其实硬盘是很复杂的，比如如何初始化、各种状态的意义，总之我当初看 AT 手册的时候就被吓到了，有兴趣的同学可以下载 AT_Attachment_with_Packet_Interface，该手册一共 3 卷。

表 3-17 中列出的端口号，是咱们今后使用硬盘时的端口范围，虽然已经精简了太多，但看上去仍然有点小畏惧，一口吃不成胖子，下面各端口何意，容我慢慢道来。

表 3-17　　硬盘控制器主要端口寄存器

IO 端口		端口用途	
Primary 通道	Secondary 通道	读操作时	写操作时
Command Block registers			
0x1F0	0x170	Data	Data
0x1F1	0x171	Error	Features
0x1F2	0x172	Sector count	Sector count
0x1F3	0x173	LBA low	LBA low
0x1F4	0x174	LBA mid	LBA mid
0x1F5	0x175	LBA high	LBA high
0x1F6	0x176	Device	device
0x1F7	0x177	Status	Command
Control Block registers			
0x3F6	0x376	Alternate status	Device Control

端口可以被分为两组，Command Block registers 和 Control Block registers。Command Block registers 用于向硬盘驱动器写入命令字或者从硬盘控制器获得硬盘状态，Control Block registers 用于控制硬盘工作状态。在 Control Block registers 组中的寄存器已经精减了，而且咱们基本上用不到，不说它们了，下面重点介绍 Command Block registers 组中的寄存器。

端口是按照通道给出的，也就是说，大家不要像我当初那样误以为端口是直接针对某块硬盘的，不是这样的，一个通道上的主、从两块硬盘都用这些端口号。要想操作某通道上的某块硬盘，需要单独指定。瞧，上面有个叫 device 的寄存器，顾名思义，指的就是驱动器设备，也就是和硬盘相关。不过此寄存器是 8 位的，一个通道上就两块硬盘，指定哪一个硬盘只用 1 位就够了，寄存器可是很宝贝的资源，不能浪费，所以此寄存器是个杂项，很多设置都需集中在此寄存器中了，其中的第 4 位，便是指定通道上的主或从硬盘，0 为主盘，1 为从盘。一会把 device 寄存器示意图拿出来给大家看看就知道了。

端口用途在读硬盘和写硬盘时还是有点区别的，比如拿 Primary 通道上的 0x1F1 端口来说，读操作时，若读取失败，里面存储的是失败状态信息，所以称为 error 寄存器，并且 0x1F2 端口中存储未读的扇区数。写操作时就变成了 feauture 寄存器，此寄存器用于写命令的参数。有没有人这样想，不就是这一个区别吗，再加个寄存器不就行了吗，何必还分情况来表示两种用途？首先，很久之前寄存器成本很贵，即使在今天也不便宜，为了节约成本，人是什么都干得出来的，哈哈，这么说有点吓人啦，总之为了省钱算是挖空了心思，充分利用了有限的资源。不光硬盘控制器是这样，显卡也是这样呢，以后用到的时候大家就明白了。其次，为了兼容。兼容是推动发展的根基，由于历史原因就被设计成这样了，现在虽然不存在当时的问题，但若弃之不理，就意味着抛弃之前的产业链，客户也不会买单的，所以为了自保，必须得兼容之前的老产

品。在 CPU 业界，为什么 Intel 能发展到今天一派繁荣，就是因为人家每次在设计新产品时就要考虑兼容老产品，这样便可以将过去创造的财富保留，而不是推翻重来。

按照表 3-17 我们来逐一介绍下寄存器的作用。

data 寄存器在名字上我们就知道它是负责管理数据的，它相当于数据的门，数据能进，也能出，所以其作用是读取或写入数据。数据的读写还是越快越好，所以此寄存器较其他寄存器宽一些，16 位（已经很不错了，表中其他寄存器都是 8 位的）。在读硬盘时，硬盘准备好的数据后，硬盘控制器将其放在内部的缓冲区中，不断读此寄存器便是读出缓冲区中的全部数据。在写硬盘时，我们要把数据源源不断地输送到此端口，数据便被存入缓冲区里，硬盘控制器发现这个缓冲区中有数据了，便将此处的数据写入相应的扇区中。

读硬盘时，端口 0x171 或 0x1F1 的寄存器名字叫 Error 寄存器，只在读取硬盘失败时有用，里面才会记录失败的信息，尚未读取的扇区数在 Sector count 寄存器中。在写硬盘时，此寄存器有了别的用途，所以有了新的名字，叫 Feature 寄存器。有些命令需要指定额外参数，这些参数就写在 Fea ture 寄存器中。强调一下，error 和 feature 这两个名字指的是同一个寄存器，只是因为不同环境下有不同的用途，为了区别这两种用途，所以在相应环境下有不同的名字。这两个寄存器都是 8 位宽度。

Sector count 寄存器用来指定待读取或待写入的扇区数。硬盘每完成一个扇区，就会将此寄存器的值减 1，所以如果中间失败了，此寄存器中的值便是尚未完成的扇区。这是 8 位寄存器，最大值为 255，若指定为 0，则表示要操作 256 个扇区。

硬盘中的扇区在物理上是用“柱面-磁头-扇区”来定位的（Cylinder Head Sector），简称为 CHS，但每次我们要事先算出扇区是在哪个盘面，哪个柱面上，这太麻烦了，这对于磁头来说很直观，它就是根据这些信息来定位扇区的。可是咱们还是希望有一套对人来说较直观的寻址方法，我们希望磁盘中扇区从 0 开始依次递增编号，不用考虑扇区所在的物理结构。其实我在描述需求时已经说出了 LBA 的定义，这是一种逻辑上为扇区址的方法，全称为逻辑块地址（Logical Block Address）。

LBA 有两种，一种是 LBA28，用 28 位比特来描述一个扇区的地址。最大寻址范围是 2 的 28 次方等于 268435456 个扇区，每个扇区是 512 字节，最大支持 128GB。我们这里为图简单，采用 LBA28 模式。由于 128GB 已经不能满足日益增长的存储需求，硬盘越来越大了，得有相匹配的寻址方法与之配套，于是要介绍的另外一种是 LBA48，用 48 位比特来描述一个扇区的地址，最大可寻址范围是 2 的 48 次方，等于 281474976710656 个扇区，乘以 512 字节后，最大支持 131072TB，即 128PB。话说我曾经运维过的 Hadoop 集群才 14PB，看样子 LBA48 还是能撑一阵子的。

介绍完了 LBA，现在可以说 LBA 寄存器了，这里有 LBA low、LBA mid、LBA high 三个，它们三个都是 8 位宽度的。LBA low 寄存器用来存储 28 位地址的第 0～7 位，LBA mid 寄存器用来存储第 8～15 位，LBA high 寄存器存储第 16～23 位。哎？三个 8 位的加起来才 24 位，连 LBA28 都不够，咱们怎么用呢？有问题就有解决方案，这就引出了下一个寄存器，device 寄存器。

在之前说过了，device 寄存器是个杂项，它的宽度是 8 位。在此寄存器的低 4 位用来存储 LBA 地址的第 24～27 位。结合上面的三个 LBA 寄存器。第 4 位用来指定通道上的主盘或从盘，0 代表主盘，1 代表从盘。第 6 位用来设置是否启用 LBA 方式，1 代表启用 LBA 模式，0 代表启用 CHS 模式。另外的两位：第 5 位和第 7 位是固定为 1 的，称为 MBS 位，大家不用关注啦。

在读硬盘时，端口 0x1F7 或 0x177 的寄存器名称是 Status，它是 8 位宽度的寄存器，用来给出硬盘的状态信息。第 0 位是 ERR 位，如果此位为 1，表示命令出错了，具体原因可见 error 寄存器。第 3 位是 data request 位，如果此位为 1，表示硬盘已经把数据准备好了，主机现在可以把数据读出来。第 6 位是 DRDY，表示硬盘就绪，此位是在对硬盘诊断时用的，表示硬盘检测正常，可以继续执行一些命令。第 7 位是 BSY 位，表示硬盘是否繁忙，如果为 1 表示硬盘正忙着，此寄存器中的其他位都无效。另外的 4 位暂不关注。

在写硬盘时，端口 0x1F7 或 0x177 的寄存器名称是 command，和上面说过的 error 和 feature 寄存器情况一样，只是用途变了，所以换了个名字表示新的用途，它和 status 寄存器是同一个。此寄存器用来存储让硬盘执行的命令，只要把命令写进此寄存器，硬盘就开始工作了。在咱们的系统中，主要使用了三个命令。

（1）identify：0xEC，即硬盘识别。

（2）read sector：0x20，即读扇区。

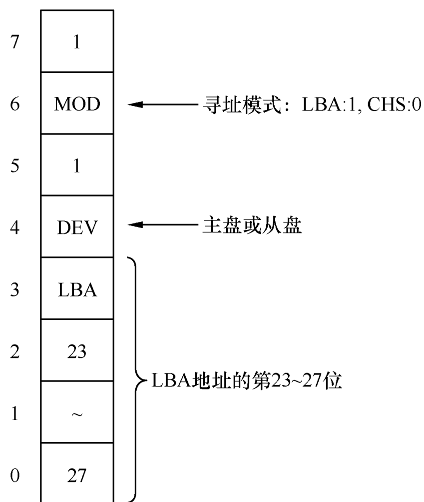
（3）write sector：0x30，即写扇区。

在此只列出了本书需要的指令，大家若对此感兴趣，还是去看 ATA 手册，您懂的，里面内容丰富详实，相信大家一定会一饱眼福。

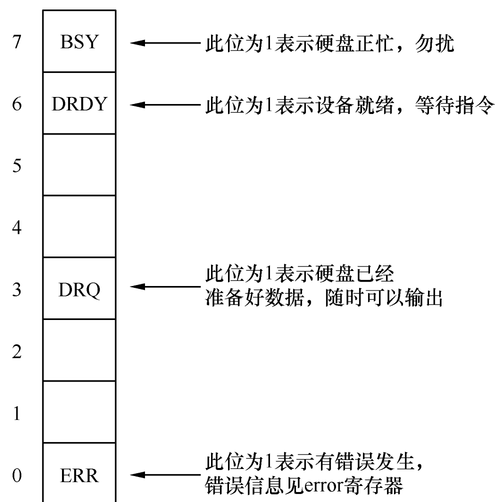
总结下寄存器 error、feature 和 status、command，大家可以这样来助记：这两组都是同一寄存器（也就是同一端口）多个用途，对同一端口写操作时，硬盘控制器认为这是个命令，对同一端口读操作时，硬盘控制器认为是想获得状态。

下面给大家呈上常用的寄存器（端口）示意图，首先出场的选手是 device 寄存器，如图 3-31 所示。

下一位选手是 status 寄存器，如图 3-32 所示。



▲图 3-31　device 寄存器



▲图 3-32　status 寄存器

3.5.4　常用的硬盘操作方法

硬盘中的指令很多，各指令的用法也不同。有的指令直接往 command 寄存器中写就行了，有的还要在 feature 寄存器中写入参数，最权威的方法还是要去参考 ATA 手册。由于本书中用到的都是简单的指令，所以对此抽象出一些公共的步骤仅供参考之用。

不管是读硬盘，还是写硬盘，都不是一个指令就完事的。相关寄存器都需要设置。要是读硬盘，得告诉读哪个扇区，读几个扇区，用哪种模式对扇区寻址，LBA？CHS？写硬盘也一样，写哪个，写几个，还要设置操作的是哪个通道的哪个硬盘……讲了这么多寄存器，心想，我到底先设置哪个寄存器呢？有没有个一般硬盘操作的基本顺序呢？还真有，小弟马上给大家呈上大概步骤。最主要的顺序就是 command 寄存器一定得是最后写，因为一旦 command 寄存器被写入后，硬盘就开始干活啦，它才不管其他寄存器中的值对不对，一律拿来就用，有问题的话报错就好啦。其他寄存器顺序不是很重要。

那咱们可以约定个操作顺序，免得大家感到无所适从，请原谅我这么说，因为我就有选择恐惧症，我很理解像我这样的同学。咱们还是约定个步骤好。

（1）先选择通道，往该通道的 sector count 寄存器中写入待操作的扇区数。

（2）往该通道上的三个 LBA 寄存器写入扇区起始地址的低 24 位。

（3）往 device 寄存器中写入 LBA 地址的 24～27 位，并置第 6 位为 1，使其为 LBA 模式，设置第 4 位，选择操作的硬盘（master 硬盘或 slave 硬盘）。

（4）往该通道上的 command 寄存器写入操作命令。

（5）读取该通道上的 status 寄存器，判断硬盘工作是否完成。

（6）如果以上步骤是读硬盘，进入下一个步骤。否则，完工。

（7）将硬盘数据读出。

硬盘工作完成后，它已经准备好了数据，咱们该怎么获取呢？一般常用的数据传送方式如下。

（1）无条件传送方式。

（2）查询传送方式。

（3）中断传送方式。

（4）直接存储器存取方式（DMA）。

（5）I/O 处理机传送方式。

对于上面的第 1 种“无条件传送方式”，应用此方式的数据源设备一定是随时准备好了数据，CPU 随时取随时拿都没问题，如寄存器、内存就是类似这样的设备，CPU 取数据时不用提前打招呼。

第 2 种“查询传送方式”，也称为程序 I/O、PIO（Programming Input/Output Model），是指传输之前，由程序先去检测设备的状态。数据源设备在一定的条件下才能传送数据，这类设备通常是低速设备，比 CPU 慢很多。CPU 需要数据时，先检查该设备的状态，如果状态为“准备好了可以发送”，CPU 再去获取数据。硬盘有 status 寄存器，里面保存了工作状态，所以对硬盘可以用此方式来获取数据。

第 3 种“中断传送方式”，也称为中断驱动 I/O。上面提到的“查询传送方式”有这样的缺陷，由于 CPU 需要不断查询设备状态，所以意味着只有最后一刻的查询才是有意义的，之前的查询都是发生在数据尚未准备好的时间段里，所以说效率不高，仅对于不要求速度的系统可以采用。可以改进的地方是如果数据源设备将数据准备好后再通知 CPU 来取，这样效率就高了。通知 CPU 可以采用中断的方式，当数据源设备准备好数据后，它通过发中断来通知 CPU 来拿数据，这样避免了 CPU 花在查询上的时间，效率较高。

第 4 种“直接存储器存取方式（DMA）”。在中断传送方式中，虽然极大地提高了 CPU 的利用率，但通过中断方式来通知 CPU，CPU 就要通过压栈来保护现场，还要执行传输指令，最后还要恢复现场。似乎有同学说此方式已经很爽了，你还想怎样？哈哈，其实更爽的是一点都不要浪费 CPU 资源，不让 CPU 参与传输，完全由数据源设备和内存直接传输。CPU 直接到内存中拿数据就好了。这就是此方式中“直接”的意思。不过 DMA 是由硬件实现的，不是软件概念，所以需要 DMA 控制器才行。

第 5 种“I/O 处理机传送方式”。不知大家发现了没有，在说上面每一种的时候都把它们各自说得特别好，似乎完美不可替代了，就像电视上的广告一样，每次都把自己的产品描述得无与伦比，甚至全宇宙第一，但该公司一出新产品，就开始自曝曾经无与伦比的老一代产品的问题以突显现在产品更胜一筹。DMA 已经借助其他硬件了，CPU 已经很轻松了，难道还有更爽的方式？是啊，DMA 方式中 CPU 还嫌爽的不够，毕竟数据输入之后或输出之前还是有一部分工作要由 CPU 来完成的，如数据交换、组合、校验等。如果 DMA 控制器再强大一点，把这些工作帮 CPU 做了就好了。也是哦，既然为了解放 CPU，都已经引用一个硬件（DMA）了，干脆一不做二不休，再引入一个硬件吧。于是，I/O 处理机诞生啦，听名字就知道它专门用于处理 IO，并且它其实是一种处理器，只不过用的是另一套擅长 IO 的指令系统，随时可以处理数据。有了 I/O 处理机的帮忙，CPU 甚至可以不知道有传输这回事，这下 CPU 才真正爽到家啦。同样，这也是需要单独的硬件来支持。

综上所述，硬盘不符合第 1 种方法，因为它需要在某种条件下才能传输。第 4 种和第 5 种需要单独的硬件支持，先不说我们的 bochs 能否模拟这两种硬件，单独学习这两类硬件的操作方法就很头疼，大家有兴趣的话还是先放一放，以后再琢磨吧。所以在我们的系统中，我们用了第 2、3 这两种软件传输方式。

关于硬盘的部分介绍完了，接下来的工作是实践，我记得当初自己做实验时的心情是非常忐忑的，总是担心有些东西不可控，有些东西自己左右不了。如果您此时的心情也是这样，那我用“过来人”的经验告诉您，想太多也没有用，做就是了，只有做超出自己能力的事才能提高，总做自己能力内的事，咱们大家连走路都不会呢。

3.6 让 MBR 使用硬盘

到目前为止，我们的 mbr 其实还没干什么正事呢。在之前的程序接力中，我们的 mbr 从 BIOS 手中接

过了这一棒。但我们的 MBR 只有 512 字节，这小小的空间，着实干不了什么大事，从 MBR 接棒的那天起，我们就知道，这一棒早晚是要交出去的。可是要交给谁呢？它在哪里？如何交给它？这一系列的问题将在本节中给出答案。

3.6.1 改造 MBR

本节我们在之前 MBR 的基础上，做个稍微大一点的改进，经过这个改进后，我们的 MBR 可以读取硬盘。听上去这可是个大“手术”呢，我们要将之前学过的知识都用上啦。其实没那么大啦，就是加了个读写磁盘的函数而已，哈哈。怀着兴奋与忐忑的心情，咱们开始吧。

改造不是乱改的，在改之前要有个计划，对将来的程序布局要有个规划，心里有数才行。先说说目前的想法。

我们的 MBR 受限于 512 字节大小的，在那么小的空间中，没法为内核准备好环境，更没法将内核成功加载到内存并运行。所以我们要在另一个程序中完成初始化环境及加载内核的任务，这个程序我们称之为 loader，即加载器。Loader 会在下一节中实现。问题来了，loader 在哪里？如何跳过去执行？这就是新款 MBR 的使命，简而言之就是，负责从硬盘上把 loader 加载到内存，并将接力棒交给它。

由于 MBR 是占据了硬盘的第 0 扇区（以逻辑 LBA 方式，扇区从 0 开始编号，若是以物理 CHS 方式，扇区则从 1 开始编号），第 1 扇区是空闲的，可以用，但离得太近总感觉不如隔开一点心里踏实，所以把 loader 放到第 2 扇区。MBR 从第 2 扇区中把它读出来。读出来放到哪里呢？原则上是找个空闲地方就行了，在表 1-1“实模式下的内存布局”中查看下，只要在“用途”列中注明“可用区域”的地方都可以用。0x500～0x7BFF 和 0x7E00～9FBFF 这两段内存区域都可以。

是这样的，容小弟分析一下。

首先，loader 中要定义一些数据结构（如 GDT 全局描述符表，不懂没关系，以后会说），这些数据结构将来的内核还是要用的，所以 loader 加载到内存后不能被覆盖。

其次，随着咱们不断添加功能，内核必然越来越大，其所在的内存地址也会向越来越高的地方发展，难免会超过可用区域的上限，咱们尽量把 loader 放在低处，多留出一些空间给内核。

所以，我将 loader 的加载地址选为 0x900。为什么不是 0x500，这个多省空间？还是预留出一定空间吧，彼此隔开远一点心里才踏实，不差这点空间了，哈哈，完全是个人偏好，大家随意啦。

按照上面所说的规划，下面代码 3-5 就是改头换面的新款 MBR。代码量增长到 126 行，下面给大家说说细节。

代码 3-5 （project/c3/b/boot/mbr.S）

```
 1 ;主引导程序
 2 ;------------------------------------------------------------
 3 %include "boot.inc"
 4 SECTION MBR vstart=0x7c00
 5    mov ax,cs
 6    mov ds,ax
 7    mov es,ax
 8    mov ss,ax
 9    mov fs,ax
10    mov sp,0x7c00
11    mov ax,0xb800
12    mov gs,ax
13
14 ;清屏
15 ;利用 0x06 号功能，上卷全部行，则可清屏
16 ; -----------------------------------------------------------
17 ;INT 0x10   功能号：0x06    功能描述：上卷窗口
18 ;------------------------------------------------------
19 ;输入：
20 ;AH 功能号= 0x06
21 ;AL = 上卷的行数(如果为 0，表示全部)
22 ;BH = 上卷行属性
23 ;(CL,CH) = 窗口左上角的(X,Y)位置
24 ;(DL,DH) = 窗口右下角的(X,Y)位置
25 ;无返回值：
```

```
26    mov    ax, 0600h
27    mov    bx, 0700h
28    mov    cx, 0           ; 左上角: (0, 0)
29    mov    dx, 184fh        ; 右下角: (80,25),
30  ; 因为 VGA 文本模式中，一行只能容纳 80 个字符，共 25 行
31             ; 下标从 0 开始，所以 0x18=24，0x4f=79
32    int    10h          ; int 10h
33
34    ; 输出字符串:MBR
35    mov byte [gs:0x00],'1'
36    mov byte [gs:0x01],0xA4
37
38    mov byte [gs:0x02],' '
39    mov byte [gs:0x03],0xA4
40
41    mov byte [gs:0x04],'M'
42    mov byte [gs:0x05],0xA4
;A 表示绿色背景闪烁,4 表示前景色为红色
43
44    mov byte [gs:0x06],'B'
45    mov byte [gs:0x07],0xA4
46
47    mov byte [gs:0x08],'R'
48    mov byte [gs:0x09],0xA4
49
50    mov eax,LOADER_START_SECTOR   ; 起始扇区 lba 地址
51    mov bx,LOADER_BASE_ADDR       ; 写入的地址
52    mov cx,1                      ; 待读入的扇区数
53    call rd_disk_m_16             ; 以下读取程序的起始部分(一个扇区)
54
55    jmp LOADER_BASE_ADDR
56
57 ;--------------------------------------------------------------------------------
58 ;功能:读取硬盘 n 个扇区
59 rd_disk_m_16:
60 ;--------------------------------------------------------------------------------
61                                  ; eax=LBA 扇区号
62                                  ; bx=将数据写入的内存地址
63                                  ; cx=读入的扇区数
64       mov esi,eax        ;备份 eax
65       mov di,cx         ;备份 cx
66 ;读写硬盘:
67 ;第 1 步:设置要读取的扇区数
68       mov dx,0x1f2
69       mov al,cl
70       out dx,al          ;读取的扇区数
71
72       mov eax,esi         ;恢复 ax
73
74 ;第 2 步:将 LBA 地址存入 0x1f3 ~ 0x1f6
75
76       ;LBA 地址 7～0 位写入端口 0x1f3
77       mov dx,0x1f3
78       out dx,al
79
80       ;LBA 地址 15～8 位写入端口 0x1f4
81       mov cl,8
82       shr eax,cl
83       mov dx,0x1f4
84       out dx,al
85
86       ;LBA 地址 23～16 位写入端口 0x1f5
87       shr eax,cl
88       mov dx,0x1f5
89       out dx,al
90
91       shr eax,cl
92       and al,0x0f         ;lba 第 24～27 位
93       or al,0xe0          ; 设置 7～4 位为 1110,表示 lba 模式
94       mov dx,0x1f6
95       out dx,al
96
```

```
 97 ;第 3 步:向 0x1f7 端口写入读命令,0x20
 98       mov dx,0x1f7
 99       mov al,0x20
100       out dx,al
101
102 ;第 4 步:检测硬盘状态
103   .not_ready:
104       ;同一端口,写时表示写入命令字,读时表示读入硬盘状态
105       nop
106       in al,dx
107       and al,0x88  ;第 4 位为 1 表示硬盘控制器已准备好数据传输
                   ;第 7 位为 1 表示硬盘忙
108       cmp al,0x08
109       jnz .not_ready      ;若未准备好,继续等
110
111 ;第 5 步:从 0x1f0 端口读数据
112       mov ax, di
113       mov dx, 256
114       mul dx
115       mov cx, ax
; di 为要读取的扇区数,一个扇区有 512 字节,每次读入一个字
116                     ; 共需 di*512/2 次,所以 di*256
117       mov dx, 0x1f0
118   .go_on_read:
119       in ax,dx
120       mov [bx],ax
121       add bx,2
122       loop .go_on_read
123       ret
124
125    times 510-($-$$) db 0
126    db 0x55,0xaa
```

程序最开始的%include "boot.inc"，这个%include 是 nasm 编译器中的预处理指令，意思是让编译器在编译之前把 boot.inc 文件包含进来。任何编译器都应该有 include 之类的能够包含其他文件的预处理指令，不要认为底层的汇编语言就应该简陋到一穷二白，哈哈，这和语言是没关系的，是编译器为了开发人员方便管理代码，应该加的。boot.inc 的内容很简单，目前就两句话，文件内容如下。

```
1 ;-------------   loader 和 kernel   ----------
2 LOADER_BASE_ADDR equ 0x900
3 LOADER_START_SECTOR equ 0x2
```

boot.inc 是我们的配置文件，我们目前关于加载器的配置信息就写在里面，今后还会在此添加更多的配置信息。大家看到的这两句也是预处理命令，是 nasm 提供的宏，和 C 语言中的宏是一回事。只不过 nasm 中的语法是：宏名 equ 值，而 C 语言中的宏是由#define 指令来实现的。所以 LOADER_BASE_ADDR 和 LOADER_START_SECTOR 是两个宏名。

LOADER_BASE_ADDR 定义了 loader 在内存中的位置，MBR 要把 loader 从硬盘读入后放到此处。如前所述，它的值是 0x900，说明将来 loader 会在内存地址 0x900 处。

LOADER_START_SECTOR 定义了 loader 在硬盘上的逻辑扇区地址，即 LBA 地址。前面和大家交待过啦，它等于 0x2，说明 loader 放在了第 2 块扇区。

接下来的第 4～48 行和上一版本没区别，不用多说啦。

第 50～52 行为函数 rd_disk_m_16 传递参数。在此说明一下，汇编语言中定义的函数（或者称为例程，proc），由于汇编语言能够直接操作寄存器，所以其传递参数可以用寄存器，也可以用栈。由于 C 语言中不能直接操作寄存器，所以咱们这里体验一回用寄存器来传递参数的函数是怎样实现的。另外再说明一下，用寄存器传参数，没有固定的形式，原则上用哪个寄存器都行，只要根据实际应用，别把还有用的寄存器值给覆盖就行，如果真需要用到某个正在使用中的寄存器，只要提前把该寄存器备份好就行了，如备份到其他寄存器或压入栈中。此函数需要三个参数，我们选择用 eax、bx、cx 寄存器来传递参数。

在寄存器 eax 中的是待读入的扇区起始地址，赋值后 eax 为定义的宏 LOADER_START_ SECTOR，即 0x2。

寄存器 cx 是读入的扇区数，cx 其值为 1。到底读入几个扇区，是由实际文件大小来决定的。由于将来会写一个简单的 loader，其大小肯定不会超过 512 字节，所以此处读入的扇区数量为 1 即可。

数据从硬盘读进来后放在内存中哪里呢？这就要用寄存器 bx 来指定。在这里，bx 寄存器值为 LOADER_BASE_ADDR，即 0x900。函数名 rd_disk_m_16 的意思是“在 16 位模式下读硬盘”。此函数是咱们本节的重点，大伙儿一定要拿下。

第 64 行的“mov esi，eax”是把 eax 中的值先备份到 esi 中。因为 al 在 out 指令中会被用到，这会影响到 eax 的低 8 位。

第 65 行是备份读取的扇区数到 di 寄存器，di 寄存器是 16 位的，和 cx 大小一致。cx 的值会在读取数据时用到，所以在此提前备份。

第 67～70 行，按照咱们操作硬盘的约定，先选定一个通道，再往 sector count 寄存器中写扇区数。往端口中写入数据用 out 指令，注意 out 指令中 dx 寄存器是用来存储端口号的。其操作格式可见 3.3.1 节的结尾部分。先查看咱们 bochs 配置文件关于硬盘的配置部分，如图 3-33 所示。

```
# 硬盘设置
ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
ata0-master: type=disk, path="hd60M.img", mode=flat, cylinders=121, heads=16, spt=63
```

▲图 3-33　bochs 中硬盘的配置

咱们的虚拟硬盘属于 ata0，是 Primary 通道，所以其 sector count 寄存器是由 0x1f2 端口来访问的。顺便再看第二行的 ata0-master，path=”hd60M.img”，这说明 hd60M.img 是主盘。

第 74～95 行是将 LBA 地址写入三个 LBA 寄存器和 device 寄存器的低 4 位。端口 0x1f3 是寄存器 LBA low，端口 0x1f4 是寄存器 LBA mid，端口 0x1f5 是寄存器 LBA high。shr 指令是逻辑右移指令，这里主要通过此指令置换出地址的相应部分，写入相应的 LBA 寄存器。第 93 行的“or al，0xe0”，用了 or“或”指令和 0xe0 做或运算，拼出 device 寄存器的值。高 4 位为 e，即高 4 位的二进制表示为 1110，其第 5 位和第 7 位固定为 1，第 6 位为 1 表示启用 LBA。大家可以参考注释。

第 97～100 行便是写入命令啦，因为我们这里是读操作，所以读扇区的命令是 0x20。通过 out 指令写入 command 端口 0x1f7 后，硬盘就开始工作了。

第 102～109 行检测 status 寄存器的 BSY 位。由于 status 寄存器依然是 0x1f7 端口，所以不需要再为 dx 重新赋值。105 行的 nop 表示空操作，即什么也不做，只是为了增加延迟，相当于 sleep 了一小下，目的是减少打扰硬盘的工作。对同一端口在读写两种操作时有不同的用途，在读硬盘时，此端口中的值是硬盘的工作状态。第 106 行是将 Status 寄存器的值读入到 al 寄存器，通过第 107 行的 and“与”操作，保留第 4 位和第 7 位，第 4 位若为 1，表示数据已经准备好，可以传输了。若第 7 位为 1，表示硬盘现在正忙着。只要判断第 4 位是否为 1 就好了，用第 108 行的 cmp 指令和 0x08 做减法运算，判断第 4 位是否为 1。cmp 指令并不改变操作数的值，只是根据结果去设置标志位，从而咱们根据标志位反着去判断结果。cmp 指令会影响的标志位有 ZF、CF、PF 等，这里咱们借助 ZF 位来判断 cmp 的结果。于是用第 109 行的 jnz .not_ready 来判断结果是否不等于 0，即若等于 0，则 status 寄存器的第 4 位为 1，这表示可以读数据了。若不等于 0，说明 status 寄存器的第 4 位为 0，表示硬盘正忙（此时 status 寄存器第 7 位肯定为 1）。.not_ready 是个标号，于是跳回去继续判断硬盘状态，直到硬盘把数据准备好才跳出这个循环。

第 111～122 行是从硬盘取数据的过程。由于 data 寄存器是 16 位，即每次 in 操作只读入 2 字节，根据读入的数据总量（扇区数*512 字节）来求得执行 in 指令的次数。这里的乘法用 mul 指令，在实模式下，mul 指令可以做 8 位乘法和 16 位乘法，格式是：mul 操作数。操作数可以是寄存器或内存。乘法运算至少要有两个数参与才行，这里的操作数只是一个乘数，被乘数隐含在 al 或 ax 寄存器中（mul 指令被设计成这样的，由于历史原因产生很多奇怪的用法，习惯就好啦）。如果操作数是 8 位，被乘数就是 al 寄存器的值，乘积就是 16 位，位于 ax 寄存器。如果操作数是 16 位，被乘数就是 ax 寄存器的值，乘积就是 32 位，积的高 16 位在 dx 寄存器，积的低 16 位在 ax 寄存器。

虽然我们进行的是 16 位的乘法，其结果是 32 位，但由于我知道这两个乘数 ax 的值和 dx 的值都不大，

ax 的实际的值其实是 1，乘出来的这个结果，其高位是 0，所以在第 115 行的“mov cx，　ax”我们只将这个结果的低 16 位移入 cx 作为循环读取的次数。此处用 8 位乘法不合适，因为 256 超过了 8 位寄存器表示的范围。在第 118～122 行通过循环来将数据写入 bx 寄存器指向的内存，每读入 2 个字节，bx 所指的地址便+2。值得注意的是由于在实模式下偏移地址为 16 位，所以用 bx 只会访问到 0～FFFFh 的偏移。待写入的地址超过 bx 的范围时，从硬盘上读出的数据会把 0x0000～0xffff 的覆盖，所以此处加载的程序不能超过 64KB，即 2 的 16 次方等于 65536。由于本 mbr 是用来加载 loader 的，所以 loader.bin 要小于 64KB 才行。这一点大可以放心，我们最终的 loader 不超过 2KB，将来的内核也不会超过 70KB。

也许有同学会说，把 bx 改为 ebx 行吗？也不行，在实模式下，CPU 依然会用 16 位偏移地址。这是实模式下访问内存的规定与缺陷，还记得那个“段基址+段内偏移地址”吗？段内偏移地址正因为是 16 位，只能访问 64KB 的段空间，所以才将段基址乘以 16 来突破这 64KB，从而实现访问低调 1MB 空间的。

第 123 行返回指令 ret，它用来从函数中返回。如果我们没有定义函数，就不需要它了。函数和一般代码相比，就是在被调用时，CPU 会将返回地址压到栈中，所以在函数体中，要用 ret 指令将栈中的返回地址重新加载到程序计数器中，如 cs：ip，这样程序便恢复到之前的执行顺序了。

执行完第 123 行后，程序便回到了第 55 行，这是个跳转的指令。个人觉得，jmp 指令和 call 指令是必不可少的，jmp 表示一去不回头，call 表示去了还回来。各有各的用途。这里是 MBR 交出接力棒的一刻，采用 jmp 是唯一合适的选择。Jmp 的操作数是 LOADER_BASE_ADDR，即 0x900，这是要跳到内核加载器的节奏。MBR 到此结束了使命，顺序完成了第二棒的拼接。复习一下，第一棒是谁来着？是 BIOS 交给了 MBR。

接下来的工作是编译，本次的编译较之前相比，多加了一个参数 -I。此参数的意思还是参见 nasm 帮助，nasm –h 回车，找到-I 的说明。

“-I<path>　　adds a pathname to the include file path”

大概意思是添加一个包含文件的路径，其实就是添加个库目录。为了目录整洁一些，我在 boot 目录下建立了一个子目录 include，并把 boot.inc 放到了 include 目录下。nasm 要用-I 指定库目录，所以在 boot 目录下输入：

nasm -I include/ -o mbr.bin mbr.S 回车

接下来用 dd 命令将 mbr.bin 写入虚拟硬盘：dd if=./mbr.bin of=/此处替换成你的安装目录/bochs/hd60M.img bs=512 count=1　conv=notrunc 回车，下面是 dd 命令的三行输出。

记录了 1+0 的读入

记录了 1+0 的写出

512 字节（512 B）已复制，0.0265972 秒，19.3 kB/秒

dd 命令输出的第三行显示了实际写入硬盘的数据大小，是 512 字节。

现在还没有准备好 loader，所以目前不宜执行。如果好奇心实在太大了，可以运行一下试试，反正只是虚拟机，对物理机不会有伤害，也许会 CPU 使用率过高。记得用 Ctrl+C 在 bochs 控制台中断运行就好了。

说了半天咱们还没有 loader 呢，若此时执行此 MBR，CPU 会直接跳到 0x900 的地方，非乱了不可，程序的运行不可预测。难为大家一直跟我在这假想这个虚幻的 loader，下一节我们要实现个真的 loader 啦。

MBR 大致就说到这，大家若是不理解，也不要糊弄自己，还是建议大家一行一行地看，直到弄清楚为止。代码写得不完美，请大家多多包涵。

3.6.2　实现内核加载器

这一节的内容并不长，因为在进入保护模式之前，我们能做的不多，loader 是要经过实模式到保护模式的过渡，并最终在保护模式下加载内核。本节只实现一个简单的 loader，本 loader 只在实模式下工作，等学习了保护模式后，我们再来个真格的。

由于本节较容易，没有新知识，直接上菜啦，见代码 3-6。

代码 3-6　（project/c3/b/boot/loader.S）

```
1 %include "boot.inc"
2 section loader vstart=LOADER_BASE_ADDR
```

```
 3
 4 ; 输出背景色绿色,前景色红色,并且跳动的字符串"1 MBR"
 5 mov byte [gs:0x00],'2'
 6 mov byte [gs:0x01],0xA4     ; A 表示绿色背景闪烁，4 表示前景色为红色
 7
 8 mov byte [gs:0x02],' '
 9 mov byte [gs:0x03],0xA4
10
11 mov byte [gs:0x04],'L'
12 mov byte [gs:0x05],0xA4
13
14 mov byte [gs:0x06],'O'
15 mov byte [gs:0x07],0xA4
16
17 mov byte [gs:0x08],'A'
18 mov byte [gs:0x09],0xA4
19
20 mov byte [gs:0x0a],'D'
21 mov byte [gs:0x0b],0xA4
22
23 mov byte [gs:0x0c],'E'
24 mov byte [gs:0x0d],0xA4
25
26 mov byte [gs:0x0e],'R'
27 mov byte [gs:0x0f],0xA4
28
29 jmp $                ; 通过死循环使程序悬停在此
```

对这个 loader 中的代码，大家是否觉得好亲切、毫无压力呢？这和咱们最初的那个 MBR 好接近，不同的是在这个 loader 中，打印的字符串是“2 loader”。

本 loader 程序第 2 行代码用到了 LOADER_BASE_ADDR，所以在第 1 行中把 boot.inc 包含进来了，其值是 0x900。其他代码就不用讲啦。

编译 nasm -I include/ -o loader.bin loader.S 回车

将生成的 loader.bin 写入硬盘第 2 个扇区。第 0 个扇区是 MBR，第 1 个扇区是空的未使用，原因如前所述，纯粹个人喜好。

dd if=./loader.bin of=/此处替换成你的安装目录/bochs/hd60M.img bs=512 count=1 seek=2 conv=notrunc 回车，下面是 dd 命令的三行输出。

记录了 0+1 的读入

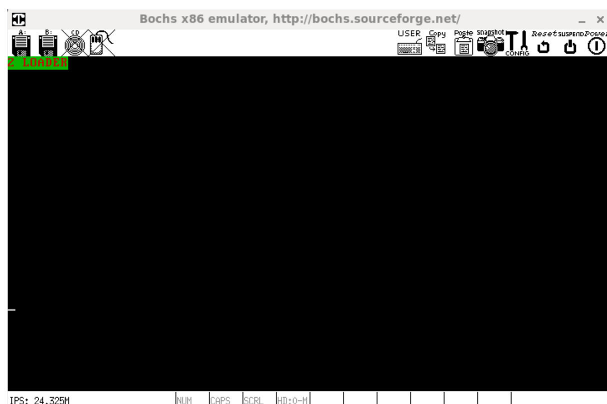
记录了 0+1 的写出

98 字节（98 B）已复制，8.9113e-05 秒，1.1 MB/s

可见，我们的 loader.bin 只有 98 字节，远远小于 64KB。

小激动的时刻到了，我们该运行 bochs 来验证了。如果程序正确的话，MBR 会跳转到 loader.bin 去运行，屏幕上会显示“2 loader”。

启动虚拟机，执行。效果如图 3-34 所示。



▲图 3-34 loader 运行状态

这次我只抓了一张图，但我人格保证这是跳动的字符，大家在自己的虚拟机上体验体验吧。

Loader 刚刚开了个头，马上就要和大家暂别了。因为这个 loader 目前还没有实际意义，目前只是来验证 MBR 和 loader 的接力是否成功，它最终的任务是要加载内核。可是内核运行在 32 位保护模式环境下，我们当前还在实模式下呢。首先咱们得知道什么是保护模式，其次还得想办法进入到保护模式，前面的路还很远。

好啦，本章到此告一段落，等我们学习保护模式后，我们还会回来继续改进 loader。

第 4 章　保护模式入门

在第 3 章里我们介绍实模式的时候，有提过一两句保护模式，而且我们也说了下实模式存在的问题。到了本章，问题便在这里结束了，我们看看 CPU 工程师是怎样解决这些问题的。随后我们要把内核加载器载入内存，体验一把保护模式带来的爽快。

下面欢迎保护模式登场。

4.1 保护模式概述

在保护模式下，我们将见到很多在实模式下没有的新概念，很多都是 CPU 硬件原生提供，并且要求的东西，也就是说按照 CPU 的设计，必须有这些东西 CPU 才能运行。咱们只要了解它们是什么并且怎么用就行了，不用深入到硬件之中挖掘其工作原理，咱们虽然是在做底层开发，但依然是应用型开发，我们能开发出什么样的软件，取决于 CPU 给咱们提供什么样的功能。就像能做出什么样的动画，主要取决于动画设计软件的功能是否强大，动画设计师的聪明才智虽然是无限的，但其驾驭能力还是很大程度上受限于软件本身的功能。今后大家会见到全局描述符表、中断描述符表、各种门结构等，所以，大家不用对这些硬件相关的概念感到慌乱，这些是 CPU 提供给咱们应用的，咱们用好就行了，还记得之前说过的公设吗？这也是。

保护模式强调的是“保护”，它是在 Intel 80286 CPU 中首次出现的，这是继 8086 之后，Intel 紧接着推出的一款产品。可见，当时的 8086 是多么地缺乏安全感。

保护模式中的“保护”体现在哪里？怎么就安全了？这个问题就像幸福的人问什么是幸福一样，还记得以前在百度工作时，我的技术经理李智勇问我：“你幸福吗？”这时候我的项目经理贺亚涛抢答：“什么是幸福？”哈哈，玩笑，您懂的。

“想要啥就有啥”并不是真正的幸福，而是发自内心地感恩、珍惜目前所拥有的一切。其实咱们已经被保护很久了，所以咱们才可以自由自在地享受计算机带来的便利。

4.1.1 为什么要有保护模式

幸福是比出来的，这一点不假。让我们看看 CPU 实模式的不幸，大家就清楚保护模式的幸福了。

（1）实模式下操作系统和用户程序属于同一特权级，这哥俩平起平坐，没有区别对待。

（2）用户程序所引用的地址都是指向真实的物理地址，也就是说逻辑地址等于物理地址，实实在在地指哪打哪。

（3）用户程序可以自由修改段基址，可以不亦乐乎地访问所有内存，没人拦得住。

以上 3 个原因属于安全缺陷，没有安全可言的 CPU 注定是不可依赖的，这从基因上决定了用户程序乃至操作系统的数据都可以被随意地删改，一旦出事往往都是灾难性的，而且不容易排查。

（4）访问超过 64KB 的内存区域时要切换段基址，转来转去容易晕乎。

（5）一次只能运行一个程序，无法充分利用计算机资源。

（6）共 20 条地址线，最大可用内存为 1MB，这即使在 20 年前也不够用。

第（4）、（5）条是使用方面的缺陷，似乎当时（20 年前）还可以忍受，但第（6）条简直就是硬伤，随着计算机事业的发展，程序对内存的需求必然是越来越大，如果还是 1MB 内存，真地太束手束脚。

为了克服这种低劣的内存管理方式，处理器厂商开发出保护模式。这样，物理内存地址不能直接被程序访问，程序内部的地址（虚拟地址）需要被转化为物理地址后再去访问，程序对此一无所知。顺便说一

句，地址转换是由处理器和操作系统共同协作完成的，处理器在硬件上提供地址转换部件，操作系统提供转换过程中所需要的页表。

4.1.2　实模式不是 32 位 CPU，变成了 16 位

32 位 CPU 具有保护模式和实模式两种运行模式，可以兼容实模式下的程序。兼容实模式，是指能够正确处理好实模式下的程序，并不是说在实模式下运行时就完全变成了纯 16 位的 CPU。就像中学生做小学生的题一样，可以用小学生的知识方法来做，但并不要求自己退化成小学生的知识水平。如果不强调方法，甚至可以用中学知识来解小学生问题。

我发现部分同学可能对实模式的理解不太清楚，特专开此节予以澄清。

当年 CPU 在以 8086 为首的 16 位天下时，根本没有实模式的概念，它们不觉得自己习惯已久的模式（运行环境、运行方式）还需要被命名。直到 CPU 发展到了 32 位，新的运行模式和之前不同了，但不管怎样发展，CPU 一定要以兼容为大，它还得兼容之前 16 位的运行模式。也就是说，32 位的 CPU 具备两种运行模式，为区别这两种模式，根据之前 8086 的 16 位模式特性，将其称为实模式，为突显现在新模式的优势，称新模式为保护模式。可以这么理解，实模式是在有 32 位 CPU 时才提出的，它是 32 位 CPU 的概念，和纯粹的 16 位 CPU，如 8086 等无关。

实模式的 CPU 运行环境 16 位，保护模式的运行环境是 32 位。

当它以 16 位的实模式运行时，不是说变成纯粹的 16 位的 CPU 了（硬件不会变身），32 位 CPU 在 16 位的实模式下运行时，虽说相当于更为强大的 16 位的 CPU，但其本质可是 32 位的，这是天生的能力。所以，当它在 16 位的实模式中，依然具备处理 32 位操作数的能力。就像一个厨师既会做中餐，又会做西餐，当他做西餐时，他还是可以用中餐的厨具来烹饪西餐的，因为他两种工具都会用。厨师关注的重点是只要把西餐做出来就行。

同样对操作数来说，纯粹的 16 位 CPU，其操作数默认也是 16 位。不是说它不想处理 32 位的操作数，而是它不能，纯粹 16 位的 CPU 不具备处理 32 位数据的能力。32 位的 CPU 本身具备处理 32 位数据的能力，现在我们所说的情况是当它处于 16 位的实模式下时，不是完全退化到 16 位 CPU 了，它依然可以具备 32 位数据处理能力，就像刚才说过厨师的例子一样，CPU 也是两种都会。所以，大家要了解，本书我们所提到的实模式，不是指纯粹的 16 位的 CPU，这种纯粹的 16 位 CPU 没有实模式之说，甚至，它一直就不知道自己的运行模式居然被后者起了个名字。

所以，再次强调，我们说实模式时，指的是 32 位的 CPU 运行在 16 位模式下的状态，不是 CPU 变身成纯粹的 16 位啦，大家不要感到迷惑。

您想，开机时，32 位的 CPU 是先处于实模式，之后再进入保护模式的。如果它处于实模式时，是和 8086 等 16 位的 CPU 完全一样，是个纯粹的 16 位 CPU，那么它该如何进入到 32 位的保护模式呢？纯粹的 16 位 CPU 可不知道什么是保护模式啊。由此可见，实模式是 32 位 CPU 中的概念，指 32 位的 CPU 处于 16 位运行模式下的状态，其本质上还是 32 位的 CPU，就像大学生去做小学生的题一样，无非是大马拉小车了。

4.2　初见保护模式

前面说过啦，由于计算机发展、进化到了更强大的阶段后，为了区别之前的“远古时代”，将之前的阶段称为实模式，为了突显现阶段的“安全”优势，称现阶段为保护模式。划时代往往出现在巨大的变革之后，由于变革带来的巨大优势，从此一笔和从前划开了界限。

计算机是如何进化的？保护模式的哪些方面值得称为“进化”？让我们参观一下保护模式的世界吧。

4.2.1　保护模式之寄存器扩展

计算机无论怎么发展，向下兼容，这都是最起码的要求。原来 16 位的寄存器要兼容，访存方式也要兼容，指令格式等都要兼容才行，否则，新产品肯定没人愿意用。没人用的产品，无论如何优秀，都将成为最大的失败。

CPU 发展到 32 位后，地址总线和数据总线也发展到 32 位，其寻址空间更达到了 2 的 32 次方，4GB。内存寻址空间上去了，内存寻址方式还得兼容老办法，即“段基址：段内偏移地址”，寄存器可以用来指定段内偏移地址，还是 16 位的话，如何承担 4GB 寻址的重任？所以，寄存器宽度也要跟上才行。

也许有同学会说，寄存器不用变，还是 16 位就行，还是按照老思路段基址再乘以一个数呗。

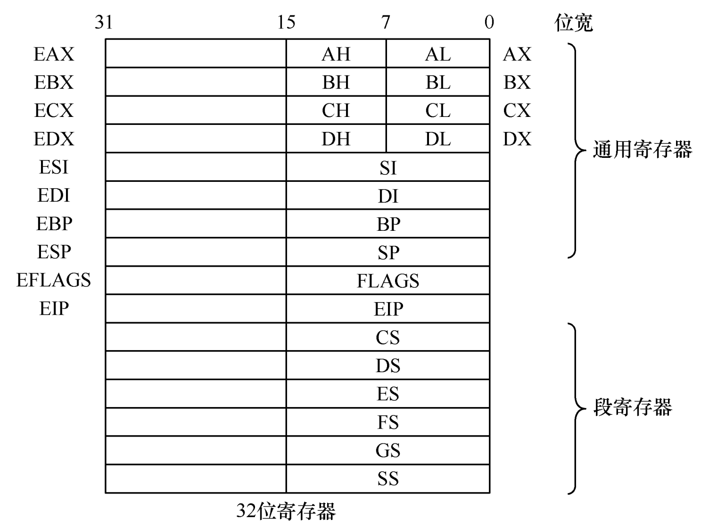
这个方案行倒是行，但段基址可不能再左移 4 位啦，得左移 16 位才行，注意是左移 16 位，不是乘以 16，要是变成乘法，得乘以 65536，即 2 的 16 次方。这样才能拼凑出 32 位地址，才能访问 32 位的地址空间。

不过呢，按理说，段基址就应该是内存段的起始地址，不应该先经过处理才能用。之前实模式下的段其址之所以先要乘以 16，那是因为单独的一个 16 位寄存器无法访问全部的 1MB 空间，限于当时的 CPU 已经成型了，为了避免“推翻重做”的重大损失，修改一下电路，悄悄地在背后给段基址乘以 16，这起码就能用了，相当于给 CPU 打个硬件补丁，就是传说中的 patch，但这毕竟只是一种亡羊补牢的作法。到了 32 位的天下，应该洗心革面，重新做 CPU，彻底将此蹩脚的方式修正，而不是带着这个补丁一瘸一拐地走下去。

所以，为了让一个寄存器就能访问 4GB 空间，需要寄存器宽度提升到 32 位。

除段寄存器外，通用寄存器、指令指针寄存器、标志寄存器都由原来的 16 位扩展到了 32 位。为什么段寄存器还是 16 位？因为段寄存器用 16 位就够用了，很奇怪是吗，答案以后揭晓。

寄存器要保持向下兼容，不能推翻之前的方案从头再来，必须在原有的基础上扩展（extend），各寄存器在原有 16 位的基础上，再次向高位扩展了 16 位，成为了 32 位寄存器。经过 extend 后的寄存器，统一在名字前加了 e 表示扩展，如图 4-1 所示。



▲图 4-1　保护模式下的被扩展成 32 位寄存器

图 4-1 中，左边已经标注名字的寄存器有通用寄存器组，名字前统一加了字符 E 表示扩展，同样，EFLAGS 寄存器和 EIP 分别在 FLAGS 和 IP 基础上扩展而成。图下边的 6 个段寄存器，依然是 16 位。

寄存器中低 16 位的部分是为了兼容实模式，可以单独使用。高 16 位没办法单独使用，只能在用 32 位寄存器时才有机会用到它们。

另外，之前咱们所讲的实模式是和 CPU 8086 运行模式一样的。但在其下一个产品 80286 之后，便开始有了保护模式。保护模式中大大提高了安全性，其中很大一部分的安全就体现在了内存段的描述方面。

偏移地址还和实模式下的一样，但段基址可不是简单的一个地址的事了。为了更加安全，怎么也得多添加点约束条件才靠谱。这些“约束条件”便是对内存段的描述信息。由于信息太多了，肯定用一个寄存器是放不下了，所以专门找了个数据结构——全局描述符表。既然叫表，就说明里面有表项，表中至少有一个表项，其中每一个表项称为段描述符，其大小为 64 字节，用来描述各个内存段的起始地址、大小、权限等信息（由于本节不是在讨论全局描述符表，故这部分会在以后章节中细说）。该全局描述符表很大，所以放在了内存中，由 GDTR 寄存器指向它就行。

这样，段寄存器中保存的再也不是段基址了，里面保存的内容叫“选择子”，selector，该选择子其实就是个数，用这个数来索引全局描述符表中的段描述符，把全局描述符表当成数组，选择子就像数组下标一样。

到了这份上，两件事要和大家说清楚。

（1）段描述符是在内存中，访问内存对 CPU 来说是比较慢的动作，效率不高。

（2）段描述符的格式很奇怪，一个数据要分三个地方存，所以 CPU 要把这些七零八落的数拼合成一个完整数据也是要花时间的。

关于段描述符格式，咱们也会在讲述全局描述符表时细说，现在先收起好奇的心，我要说重点啦。

既然访问内存中的段描述符如此地耗费时间，这在 CPU 中是实在等不了的，所以，重点来啦，在 80286 的保护模式中，为了提高获取段信息的效率，对段寄存器率先应用了缓存技术，将段信息用一个寄存器来缓存，这就是段描述符缓冲寄存器（Descriptor Cache Registers）。对程序员而言它是不可见的。CPU 每次将千辛万苦获取到的内存段信息，整理成“完整的、通顺、不蹩脚”的形式后，存入段描述符缓冲寄存器，以后每次访问相同的段时，就直接读取该段寄存器对应的段描述符缓冲寄存器。

另外，虽然段描述符缓冲寄存器是保护模式下的产物，但它也可以用在实模式下。您想，在 16 位模式下内存访问时，段基址要左移 4 位后变成 20 位地址，再与段内偏移地址相加求和，最后用所求的和作为目标地址访问内存。也就是说，每次引用一个段内地址时，段基址都要先左移 4 位，这个计算量还是蛮大的。由于切换段基址并不是特别频繁，所以有必要将段基址左移 4 位后的结果缓存起来，避免重复计算。既然已经有了段描述符缓冲寄存器，虽然它是因保护模式而创造的，但并不是说它不能用在实模式下，CPU 只是兼容实模式，不管 CPU 用什么资源，只要能把实模式下的程序处理好就行啦。所以，在实模式下时，段基址左移 4 位后的结果就被放入段描述符缓冲寄存器中，以后每次引用一个段时，就直接走段描述符缓冲寄存器，直到该段寄存器被重新赋值。

既然是缓存，就一定要有个失效时间。段描述符缓冲寄存器的失效时间是多少？其实这个时间还真没“准”，原则上，只要往段寄存器中赋值，CPU 就会更新段描述符缓冲寄存器。例如，在保护模式下加载选择子（即使新选择子的值和之前段寄存器中老的选择子相同），CPU 就会重新访问全局描述符表，再将获取的段信息重新放回段描述符缓冲寄存器，或在实模式下为段寄存器赋予段基址，无论是否与之前段基址相同，段基址左移 4 位后的结果就被送入段描述符缓冲寄存器。

下面列出三种段描述符缓冲寄存器结构，如图 4-2 所示。

80286 处理器						
47~32	31	30~29	28	27~24	23~0	偏移量
Limit	P	DPL	S	Type	base	字段

80386/80486 处理器										
95~64	63~32	31~24	23	22~21	20	19~16	15	14	13~0	偏移量
limit	base	0	P	DPL	S	Type	0	D/B	0	字段

奔腾处理器									
95~79	78	77~72	71	70~69	68	67~64	63~32	31~0	偏移量
0	D/B	0	P	DPL	S	type	base	Limit	字段

▲图 4-2 段描述符缓冲寄存器结构

80286 虽然有了保护模式，但其依然是 16 位的 CPU，其通用寄存器还是 16 位宽。但其与 8086 不同的是其地址线由 20 位变为了 24 位，即寻址空间变成了 2 的 24 次方，等于 16MB 大小。

之前 8086 也是 16 位 CPU，其通用寄存器也是 16 位，它好不容易才能够访问 1MB 内存空间。同样“硬

件配置”的 80286 如何突破 1MB 空间，能够访问 16MB 的呢？大家看图 4-2 中 80286 的段描述符缓冲寄存器结构图，原因就是段描述符中，段基址用 24 位来表示，字段名称为 base 的部分就是该描述符所描述的内存段的起始地址，占用了 0～23 位。由于 80286 的地址总线是 24 位，所以此处的段基址已经符合了地址宽度，不用再左移多少位或乘以某个数了。这就保证了地址是 24 位的，所以可以访问的空间是 16MB。

不要忘记啦，IA32 体系架构的 CPU，其访存方式还是分段策略，即“段基址：段内相对偏移地址”的形式是无法改变的。上面说到的是 24 位段基址，段内相对偏移地址还是必不可少的。用于寻址的通用寄存器还是 16 位，即单独的一个寄存器还是只能访问 64KB 的空间。如果用寄存器作为段内偏移地址，段的大小还是 64KB，这就白白浪费了 24 位段基址的优势，要是想访问完整的 16MB 内存，依然要不断地变换段基址，所以 80286 只能算个鸡肋产品，弃之可惜，食之无味，所以 80286 很快就被淘汰了，取而代之的是 80386。

80286 的问题是：单独的一个寄存器无法访问到全部内存空间，也就是若用寄存器存储段内偏移地址，只能访问到 64KB 大小的段。还有另外一个问题，每次 CPU 变革的原因几乎都是地址总线宽度不够导致的，即内存需求越来越大，干脆，Intel 直接将地址线改为 32 位（毕竟 24 位的地址线也显得不伦不类），在当时那个年代，32 位的地址空间 4GB 可以算是一步到位啦。

于是，1985 年推出了首款 32 位处理器 80386，它的地址总线和寄存器都是 32 位的。结果图 4-2 中的 80286 段描述符缓冲寄存器结构的 base 部分，这是个 32 位的段基址，位于该结构的第 32～63 位。这样，段基址是 32 位，单独的一个寄存器也是 32 位，任意一个段都可以访问到 4GB 空间啦，不用再变化段基址了。甚至段基址可以是 0，光用段内偏移就可以指向 4GB 空间任意角落。这就开启了“平坦模式”的时代，大大方便了开发人员的工作。

有了保护模式，之前的实模式下的程序还得兼容，所以便有了个“过渡模式”，即虚拟 8086 模式。知道此虚拟模式为什么包含“8086”了吧，就是因为 80286 是首款具备保护模式的 CPU，而之前的 CPU 都是只有实模式，最有代表性的、应用最广的 CPU 是 8086。

综上所述，CPU 有三种模式：实模式、虚拟 8086 模式、保护模式。

4.2.2 保护模式之寻址扩展

进入保护模式后，寻址方式也有了很大进步，基址、变址、寻址变得更加灵活了。

不知道大家是否已经将实模式下的基址、变址寻址方式忘记啦？给大家简短复习下，它的形式如图 4-3 所示。

具体形式如下代码。

```
1 mov ax,[si]
2 mov ax,[di]
3 mov ax,[bx]
4 mov ax,[bx+si]
5 mov ax,[bx+si+0x1234]
6 mov ax,[bx+di]
7 mov ax,[bx+di+0x1234]
```

以上这七种方式都是在进行内存寻址，不知道各位是否看出点端倪，实模式下对于内存寻址来说，其中的基址寻址、变址寻址、基址变址寻址，这三种形式中的基址寄存器只能是 bx、bp，变址寄存器只能是 si、di，也就是说，只能用这 4 个寄存器。其中 bx 默认的段寄存器是 ds，它经常用于访问数据段，bp 默认的段寄存器是 ss，它经常用于访问栈。

总之实模式下的寄存器有固定的使命，对于寻址来说，若想用其他的寄存器，甭说 CPU 报不报错，就连编译这关都过不了，如这句代码“mov ax，[dx]”，nasm 编译器会报：error： invalid effective address。

对于寻址中的偏移量，只能是 1 个字以内的立即数，即不能超过 16 位。如果超过了 16 位，编译器会报：warning：word data exceeds bounds。

灵活是比出来的，在保护模式下，这一切都不同了，同样是内存寻址中，基址寄存器不再只是 bx、bp，而是所有 32 位的通用寄存器，变址寄存器也是一样，不再只是 si、di，而是除 esp 之外的所有 32 位通用寄存器，偏移量由实模式的 16 位变成了 32 位。并且，还可以对变址寄存器乘以一个比例因子，注意

比例因子，只能是 1、2、4、8，如图 4-4 所示。

基址寄存器　变址寄存器　16位的偏移量

{BX BP} + {SI DI} + {立即数}

▲图 4-3　实模式的内存寻址方式

基址寄存器　变址寄存器　比例因子　32位的偏移量

{eax esi ebx edi ecx ebp edx esp} + {eax esi ebx edi ecx ebp edx} X {1 2 4 8} + {立即数}

▲图 4-4　保护模式的寻址方式

具体形式如下代码。

```
1 mov eax, [eax+edx*8+0x12345678]
2 mov eax, [eax+edx*2+0x8]
3 mov eax, [ecx*4+0x1234]
```

虽然 esp 无法用作变址寄存器，但其可用于基址寄存器。所以，如下代码是正确的。

```
1 mov eax, [esp]
2 mov eax, [esp+2]
```

保护模式下有关寻址方式的变化就讲这些啦。

4.2.3　保护模式之运行模式反转

我们当前使用的 CPU 运行模式有实模式和保护模式两种。恰恰是要兼顾这两种模式，才让 CPU 设计者变得更加辛苦。

前面说过啦，CPU 处于实模式下时，并不是变成了纯粹的 16 位 CPU，它相当于 8086 的加强版，依然可以使用 32 位下的资源。也就是说，资源是共通的，无论哪种模式都可以在指令中使用它们，问题就来了，同样一句汇编代码，它总该隶属于某种模式之下，但它到底是属于实模式，还是属于保护模式呢？也许有同学说啦，“管它干吗呢，只要编译器帮我把语句编译成合适的机器码就行啦，这一切不用我操心，这是编译器的事”。其实，编译器没有那么强大，很多时候需要人为告诉编译器一些信息，编译器才知道如何生成机器码。另外，用汇编语言编程，其最大的魅力就是实时了解机器的状况，实时清楚你在做什么，实时掌控机器的一举一动，实时将自己置身于 CPU 的角色。难道您不想拥有这种掌控力吗？

兼容给 CPU 的发展带来了不少困难，想想 CPU 真是不容易啊，一方面要发展自身功能，另一方面还要使过去的一些“古董设计”继续可用，简直就是背着个大包袱爬山。前面和大家说过啦，保护模式下的寻址方式有很大进步，通用寄存器几乎都可以用于寻址，这些寻址上的差异如何让 CPU 辨认呢？底层的电路是很死板的，一种功能就对应一种电路，因此物理上的实现往往是有限的，在人看来逻辑上相同的东西，在底层硬件中却是完全不同的电路设计。

为了兼容，当初为实模式设计的指令格式，如今还得沿用这一套，如何让两种运行模式套用同样的指令格式呢？哈哈，虽然是个问号，但也没想让大家帮着思考，大家不用操心了，人家早已经解决啦。

还记得指令格式吗？前面有介绍过，其格式是：

前缀	操作码	寻址方式、操作数类型	立即数	偏移量

在这个格式中第 3 个字段用于指定寻址方式和操作数类型，在指令格式不变的情况下，为了兼容保护模式，一种方案是重新定义各寻址方式、寄存器的编码。由于保护模式中的寻址方式和操作数类型同实模式下完全不同，故相应的编码也不同。比如在实模式下，用二进制 010 表示 dx 寄存器，在保护模式下的 010 就表示 edx 寄存器（根据编码确定指令、寻址方式、寄存器，这是译码器的工作）。操作 dx 寄存器和 edx 寄存器，对于硬件来说是完全不同的，所以编译器必须明确操作对象是哪个。

在实模式下，指令和操作数都是 16 位的，但我们也说过啦，它可以使用 32 位的资源。同样在保护模式下，指令和操作数都是 32 位的，它也可以使用 16 位的资源。也就是说，在某个模式下，可以使用另一模式下的资源。

兼容性带来了好处，也带来了坏处，好处是 CPU 很强大，可以同时支持 16 位指令和 32 位指令，运行新老程序畅通无阻。但坏处就是 CPU 也不知道您想生成 16 位，还是 32 位机器码，这就是前面说过的，需要明

确告诉编译器一些信息。为此，编译器提供了伪指令 bits，用它来向编译器传达：我下面的指令都要编译成 xx 位的，因为我知道下面的代码的运行环境是 xx 模式。比如在实模式下，运行的指令都是 16 位的，所以编译器要将代码编译成 16 位的指令。在实模式下准备好了保护模式所需要的环境后，进入保护模式后的代码就应该是 32 位指令。也就是，同一段程序要经历两种模式，所以同一段程序中有两种模式的机器码。

bits 的指令格式是[bits 16]或[bits 32]。

[bits 16]是告诉编译器，下面的代码帮我编译成 16 位的机器码。

[bits 32]是告诉编译器，下面的代码帮我编译成 32 位的机器码。

“下面的代码”是哪里呢？bits 指令的范围是从当前 bits 标签直到下一个 bits 标签的范围，这个范围中的代码将被编译成相应字长的机器码。bits 外面的中括号是可以省略的，另外，在未使用 bits 指令的地方，默认是[bits 16]。

啰嗦一下，使用 bits 指令的情况是：您清楚所写的代码是运行在哪种模式下，您需要向编译器明确指出将其编译成哪种模式的机器码。也许有同学有疑惑，打开保护模式的步骤是清晰且有限的，看上去，编译器根据代码似乎能猜测出是否是打开了保护模式，当前代码所处在的模式也能猜出，应该能自动识别编译成哪类机器码。

可是，虽然人能做到这一点，但编译器却不容易做到。进入保护模式的代码却是千奇百怪的，形式可不统一。比如进入保护模式需要三个步骤。

（1）打开 A20。

（2）加载 gdt。

（3）将 cr0 的 pe 位置 1。

这三个步骤可以不顺序，也可以不连续，并且每个步骤又是由多个小步骤完成的，每个小步骤的形式又不是固定的，如果这些组合是有穷的还好，但面对如此无穷多的组合，最聪明的编译器的设计者也会束手无策。除非编译器提供一个打开保护模式的方法，并有办法强制开发人员使用，这样一来，编译器器自然就知道何时进入了保护模式。但这样也不现实，依然不能保证开发人员不自己写进入保护模式的代码。另外，由于兼容的原因，并不是说在某模式下一定都是某模式的指令。因为前面说过啦，16 位环境下可以用 32 位环境的资源，而 32 位下也可以用 16 位的资源（编译后的二进制文件中可以有两种不同的机器码）。所以编译器不容易猜出代码所在的运行模式，用 bits 指令明确指出运行模式，这是最简单省事的办法。

说得再形象也不如举例子，见代码 1bits.S。

代码 1bits.S

```
1 [bits 16]
2 mov ax, 0x1234
3 mov dx, 0x1234
4
5 [bits 32]
6 mov eax, 0x1234
7 mov edx, 0x1234
```

代码很简单，解释无需多言，表 4-1 是此代码编译后的指令，请大家过目。

表 4-1　　bits 使用

行　号	指　令	机　器　码
1	[bits 16]	伪指令，无机器码
2	mov ax, 0x1234	B83412
3	mov dx, 0x1234	BA3412
4	[bits 32]	伪指令，无机器码
5	mov eax, 0x1234	B834120000
6	mov edx, 0x1234	BA34120000

大家看表 4-1 第 2 行，在 16 位模式，mov ax, 0x1234 的机器码 B83412，第 5 行 32 位模式下 mov eax, 0x1234，其机器码是 B834120000，这两者都是同样的操作码 B8。不同的模式下都是同样的操作码，可见，

寻址方式和操作数类型相同，但意义是不同的，在这里一个是 ax 寄存器，另一个是 eax 寄存器。

由于在两种模式下，同样的操作码和操作数编码，却有着不同的解释，为了让 CPU 能够在第一时间了解要执行的是哪种模式下的指令，到底指令中的编码对应哪些寄存器，在机器码的最前面就应该存放一些用于识别的标识。这么做应该是极好的，所以在指令的最前面有个前缀字段，用来率先告诉 CPU 应用此指令的模式。

其实上段文字的铺垫，是我要在这里介绍反转前缀。反转前缀是干吗的呢？

在指令格式中，有个“前缀”字段，里面存放的是指令选项之类的东东，比如指令重复前缀 rep、段跨越前缀“段寄存器:”，还有咱们马上要介绍的操作数反转前缀 0x66 和寻址方式反转前缀 0x67。

我们已经知道了，在不同的模式下，操作数和寻址方式都各不相同。实模式下的操作数大小是 16 位，保护模式下的操作数大小是 32 位。而我们在介绍保护模式的寻址模式时，通过对比，大家也看出了实模式和保护模式在寻址方面的差别。

要说段跨越前缀这个好懂，操作数反转前缀这个是干吗的呢？我们之前其实说过 N 多次了，模式之间可以互相使用对方环境下的资源。比如，16 位实模式下可以用 32 位保护模式下的寄存器。但这种福利的得来却是稍费功夫的，如果要用另一模式下的操作数大小，需要在指令前添加指令前缀 0x66，将当前模式临时改变成另一模式。这就是反转的意义，不管是当前模式是什么，总是转变成相反的运行模式。

比如，在指令中添加了 0x66 反转前缀之后：

假设当前运行模式是 16 位实模式，操作数大小将变为 32 位。

假设当前运行模式是 32 位保护模式，操作数大小将变为 16 位。

注意啦，这个转换只是临时的，只在当前指令有效。

还是实际例子更能说明问题，见代码 2bits.S。

代码 2bits.S

```
1 [bits 16]
2 mov ax, 0x1234
3 mov eax, 0x1234
4
5 [bits 32]
6 mov ax, 0x1234
7 mov eax, 0x1234
```

表 4-2 是上述代码编译后的机器指令。

表 4-2　　0x66 反转

行　　号	指　　令	机　器　码
1	[bits 16]	伪指令，无机器码
2	mov ax, 0x1234	B83412
3	mov eax, 0x1234	66B834120000
4	[bits 32]	伪指令，无机器码
5	mov ax, 0x1234	66B83412
6	mov eax, 0x1234	B834120000

第 1 行表示让编译器将第 2、3 行编译成 16 位机器码。

第 2 行就是 16 位模式的指令，所以直接编译，机器码为 B83412。

第 3 行用到了 32 位寄存器 eax，属于 32 位操作数，由于当前模式是 16 位，要用 0x66 将操作数大小转为 32 位，故机器码是 66B834120000。其中 34120000 是另一操作数，B8 是操作码，0x66 便是操作数反转前缀。

第 4 行表示让编译器将第 4、5 行编译成 32 位机器码。

第 5 行是 16 位指令，但当前已在 32 位模式下，所以要用操作数反转前缀 0x66 来临时将当前模式的 32 位操作数反转成 16 位大小的操作数，故机器码是 66B83412。最前面的 0x66 正是反转前缀，b8、3412 分别是操作码和操作数。

第 6 行就是 32 位指令，所以符合当前模式。B8 是操作码，34120000 是操作数。

以上是反转操作数大小前缀 0x66 的应用。下面再介绍个前缀：寻址方式反转前缀 0x67。

不同模式之间不仅可以使用对方模式下的操作数，还可以使用对方模式下的寻址方式。猛一看这句话，似乎不容易接受，其实感性一点想想，32 位 CPU 两种模式都兼容，无论它处于哪个模式，都是它自己。就像孙悟空变成马时，难道它最爱吃的就是草料吗？当然不，即使变成了马，它本质还是猴子，它的最爱还是桃子。

上代码，实践出真知，见文件 0x67.S。

文件 0x67.S

```
1 [bits 16]
2 mov word [bx], 0x1234
3 mov word [eax], 0x1234
4 mov dword [eax], 0x1234
5 [bits 32]
6 mov dword [eax], 0x1234
7 mov word [eax], 0x1234
8 mov dword [bx], 0x1234
```

以上并不是每句代码都会用到反转寻址方式前缀 0x67。只有 3、4、8 这三行牵扯到寻址方式反转。其余的“正常”代码都是为了与“反转寻址”对比。

表 4-3 是上述代码编译后的机器指令。

表 4-3　　寻址方式反转前缀 0x67

行　　号	指　　令	机　器　码
1	[bits 16]	伪指令，无机器码
2	mov word [bx], 0x1234	C7073412
3	mov word [eax], 0x1234	67C7003412
4	mov dword [eax], 0x1234	6667C70034120000
5	[bits 32]	伪指令，无机器码
6	mov dword [eax], 0x1234	C70034120000
7	mov word [eax], 0x1234	66C7003412
8	mov dword [bx], 0x1234	67C70734120000

以上我们是用的内存寻址中的基址寻址，注意啦，为什么实例中用了 eax 和 bx 两种寄存器，而不是 ebx 和 bx？因为实模式下的基址只有用寄存器 bx、bp。

见表 4-3，第 1 行的[bits 16]指示编译器：把下面 2、3、4 行的代码编译成 16 位代码。

第 2 行的指令，从操作数和寻址方式来看，本身符合 16 位模式，无需添加任何反转前缀。第 3 行把 eax 寄存器作为基址寻址，eax 寄存器不属于实模式，所以在机器码前添加了寻址方式反转前缀 0x67。第 4 行同样是用 eax 寄存器作为基址寻址，并且用到了伪指令 dword，表示在 eax 所表示的内存处，连续写入 4 字节大小的数据。操作数大小也由默认的 2 字节变成了 4 字节，就会添加 0x66 的前缀。所以其机器码是 6667C70034120000，前面 2 字节是前缀 0x66、0x67。

第 5 行指示编译器，将下面指令编译成 32 位。

第 6 行的指令，无论从操作数，还是寻址方式来看，它都是纯粹的 32 位指令，所以机器码不包含任何反转前缀。

第 7 行用到了伪指令 word，这是一种数据类型，表示 2 字节。在本行代码中表示从 eax 指定的内存处连续写入 2 字节。这就改变了操作数的大小，当前已经是 32 位机器码了，默认操作数也是 32 位，所以要用操作数大小反转前缀 0x66，见机器码第 1 字节。

第 8 行的指令使用的寻址方式是实模式下的基址寻址，寄存器是 bx（只能是 bx 或 bp），但由于当前是 32 位保护模式，所以在机器码中要用寻址方式反转前缀 0x67。

关于 CPU 在两个模式下的运行原理，本节只是蜻蜓点水般地讲述了一些浅表知识。由于咱们项目代码中关于模式的部分少之又少，而且，添加指令前缀，这又是编译器的工作，所以大家有可能都不会察觉到它的存在。更为要命的是兄弟我水平实在有限，只有讲出这么多啦，所以本书关于运行模式的介绍就到这里。

总结一下，bits 伪指令用于指定运行模式，操作数大小反转前缀 0x66 和寻址方式反转前缀 0x67，用于临时将当前运行模式下的操作数大小和寻址方式转变成另外一种模式下的操作数大小及寻址方式。

以上两点便是本节的精华所在。

4.2.4　保护模式之指令扩展

在 16 位的实模式下，CPU 的操作数是 16 位。在 32 位的保护模式下，操作数扩展到了 32 位，于是涉及到操作数变化的指令也要跟着扩展，既要兼容 16 的操作数，也要支持 32 位的操作数，想想处理器的设计者真不容易啊。

比如 add，大家知道这是用于加法运算的指令，实模式下时，它的操作数可以为 8 位、16 位，在如今的保护模式中，它不仅要支持 8 位、16 位，还得支持 32 位的操作数，如：

```
1 add al, cl            ;支持 8 位操作数
2 add ax, cx            ;支持 16 位操作数
3 add eax, ecx          ;支持 32 位操作数
```

对于减法指令 sub 也是一样。

```
1 sub al,cl             ;支持 8 位操作数
2 sub ax,cx             ;支持 16 位操作数
3 sub eax,ecx           ;支持 32 位操作数
```

同一指令在不同的模式中要做出不同的行为，可见 CPU 精密的设计。

以上说的是双操作数的指令，还有一些单操作数指令，如 inc、dec 等，也是同时支持 8 位、16 位、32 位寄存器。

并不是所有的指令都要支持以上 3 种宽度的操作数，比如对于 loop 指令，实模式下要用 cx 寄存器来存储循环次数，在保护模式下，要用 ecx。

以上这些还好，至少操作数还只是在一个寄存器中。下面要说的这两个指令，为了支持 32 位操作数，不得不增加了额外的寄存器。

mul 指令是无符号数相乘指令，指令格式是 mul　寄存器/内存。

其中“寄存器/内存”是乘数。

如果乘数是 8 位，则把寄存器 al 当作另一个乘数，结果便是 16 位，存入寄存器 ax。

如果乘数是 16 位，则把寄存器 ax 当作另一个乘数，结果便是 32 位，存入寄存器 eax。

如果乘数是 32 位，则把寄存器 eax 当作另一个乘数，结果便是 64 位，存入 edx：eax，其中 edx 是积的高 32 位，eax 是积的低 32 位。

有符号数相乘指令 imul 也是一样，不再说明。

对于无符号数除法指令 div，其格式是 div 寄存器/内存，其中的“寄存器/内存”是除法计算中的除数。

如果除数是 8 位，被除数就是 16 位，位于寄存器 ax。所得的结果，商在寄存器 al，余数在寄存器 ah。

如果除数是 16 位，被除数就是 32 位，被除数的高 16 位则位于寄存器 dx，被除数的低 16 位则位于寄存器 ax。所得的结果，商在寄存器 ax，余数在寄存器 dx。

如果除数是 32 位，被除数就是 64 位，被除数的高 32 位则位于寄存器 edx，被除数的低 32 位则位于寄存器 eax，所得的结果，商在寄存器 eax，余数在寄存器 edx。

本章前面部分已经说过了，由于已经是 32 位的 CPU 了，它同时具备处理 16 位和 32 位数据的能力，当它以 16 位的模式运行时，不是说变成纯粹的 16 位的 CPU 了（纯粹 16 位 CPU 是无法进入到 32 位保护模式的），32 位 CPU 在 16 位的实模式下运行时，可以理解为 16 位 CPU 的加强版。但其本身是 32 位 CPU，它天生就具备处理 32 位数据的能力。我再次重申此内容的目的是想告诉大家，在 16 位的实模式下，CPU 照样可以处理 32 位的数据，大家不要感到奇怪。

这方面最典型的例子就是 push，这是往栈中添加数据的指令。同样的指令，在实模式和保护模式下都可以同时处理 16 位和 32 位的数据，让咱们看看 push 是怎样应对这两种局面的。

对于 push 指令，需要根据其操作数的类型，分别讨论，操作数类型如下。

（1）立即数。

（2）寄存器。

（3）内存。

下面咱们看看各方面的内容。

先看第 1 种情况，对于立即数来说，可以分别压入 8 位、16 位、32 位数据。

指令格式是：

push 8 位立即数

push 16 位立即数

push 32 位立即数

虽说可以压入 8 位立即数，但实际上，对于 CPU 来说，出于对齐的考虑，操作数要么是 16 位，要么是 32 位，所以 8 位立即数会被扩展成各模式下的默认操作数宽度，即实模式下 8 位立即数扩展成为 16 位后再入栈，保护模式下扩展成为 32 位后再入栈。

在实模式环境下：

当压入 8 位立即数时，由于实模式下默认操作数是 16 位，CPU 会将其扩展为 16 位后再将其入栈，sp-2。

当压入 16 位立即数时，CPU 会将其直接入栈，sp-2。

当压入 32 位立即数时，CPU 会将其直接入栈，sp-4。

见示例代码 16push.S。

16push.S

```
1 section loader vstart=0x900
2 mov sp, 0x900
3 push byte 0x7
4 push word 0x8
5 push dword 0x9
6 jmp $
```

表 4-4　实模式下 push 指令操作数

行号	下一条指令	下一条的指令的机器码	当前 esp 值
1	mov sp,　0x0900	bc0009	0x00007c00
2	push 0x0007	6a07	0x00000900
3	push 0x0008	6a08	0x000008fe
4	push 0x00000009	666a09	0x000008fc
5	jmp .-2	ebfe	0x000008f8

大家看表 4-4，以上信息是我在 bochs 虚拟机中摘录整理的。如果您目前还没有用过虚拟机，或者不熟悉调试的话，有必要给您大概说一下。“下一条指令”列是尚未执行的指令，是下一次要执行的指令，故“当前 esp 值”与它无关。“当前 esp 值”列是当前系统的栈指针。在本例中，第 n 行的“下一条指令”将改变第 n+1 行的“当前 esp 值”。

第 1 行的当前 esp 是 0x7c00。为了更清楚地观察到栈指针变化，准备用 mov sp,　0x900 指令将 sp 赋值为 0x900。所以，在第 2 行的“当前 esp 值”列已经更新成了 0x900。

第 2 行的下一条指令是 push 0x0007，其操作码是 0x6a，这是压入一个字的操作码。大伙儿可以对照下源文件 16push.S 的第 3 行，原本是 push byte 0x7。可见，如前所述，CPU 并不是真地压入 1 字节。byte 并不是 CPU 的指令，而是编译器提供的伪指令，它给编译器指出数据的宽度。第 3 行的“当前 esp 值”列，其值为 0x8fe，这正是 push 0x0007 对 sp 指针的影响，0x900-0x8fe=2，可见，sp 的值减了 2，即向栈中压入了 2 字节的数据。

第 3 行要执行的指令是 push 0x0008，对照源文件 16push.S 的第 4 行，push word 0x8，这是直接压入一个字，其操作码是 0x6a。此指令执行后会是什么效果呢？见第 4 行的“当前 esp 值”列，esp 值为 0x8fc，0x8fe-0x8fc=2。说明 sp 的值减了 2，即向栈中压入了 2 字节的数据。

第 4 行要执行的指令是 push 0x00000009，其机器码是 666a09，其中机器码可以拆分成三个部分，09 是操作数，6a 是操作码，66 是操作数大小反转前缀，当前是 16 位的实模式，但要操作的是 32 位环境下的操作数，所以编译器在指令前添加了指令前缀 0x66 将当前的默认操作数反转成 32 位，当然这只是临时的，仅在当前指令有效。操作数由源文件中的 0x9 变成了 32 位的 0x00000009。这分明是要压入 4 字节的节奏，其对栈指针 sp 的影响体现在第 5 行的"当前 esp 值"，为 0x8f8。0x8fc-0x8f8=4，说明 sp 的值减了 4，即向栈中压入了 4 字节的数据。

在保护模式下，同样是这些压入立即数的指令，栈指针会有怎样的变化呢？

当压入 8 位立即数时，由于保护模式下默认操作数是 32 位，CPU 将其扩展为 32 位后入栈，esp 指针减 4。

当压入 16 位立即数时，CPU 直接压入 2 字节，esp 指针减 2。

当压入 32 位立即数时，CPU 直接压入 4 字节，esp 指针减 4。

本书的特色就是理论结合实践，要上 CPU 验证一下才放心。

见示例代码 32push.S，为了避免吓到部分同学，前 46 行暂时不需要阅读。

32push.S

```
 1     %include "boot.inc"
 2     section push32_test vstart=0x900
 3     jmp loader_start
 4 gdt_addr:
 5
 6 ;构建 gdt 及其内部的描述符
 7    GDT_BASE:   dd    0x00000000
 8                dd    0x00000000
 9
10    CODE_DESC:  dd    0x0000FFFF
11                dd    DESC_CODE_HIGH4
12
13    DATA_STACK_DESC:  dd    0x0000FFFF
14                      dd    DESC_DATA_HIGH4
15
16    VIDEO_DESC: dd    0x80000008
17                dd    DESC_VIDEO_HIGH4  ; 此时 dpl 已改为 0
18
19    GDT_SIZE   equ   $ - GDT_BASE
20    GDT_LIMIT   equ   GDT_SIZE - 1
21    SELECTOR_CODE equ (0x0001<<3) + TI_GDT + RPL0
22    SELECTOR_DATA equ (0x0002<<3) + TI_GDT + RPL0
23    SELECTOR_VIDEO equ (0x0003<<3) + TI_GDT + RPL0
24
25    gdt_ptr:  dw  GDT_LIMIT
26              dd  gdt_addr
27
28    loader_start:
29
30 ;---------------   准备进入保护模式   ----------------
31 ;1 打开 A20
32 ;2 加载 gdt
33 ;3 将 cr0 的 pe 位置 1
4
35    ;-----------------  打开 A20  ----------------
36    in al,0x92
37    or al,0000_0010B
38    out 0x92,al
39
40    ;-----------------  加载 GDT  ----------------
41    lgdt [gdt_ptr]
42
43    ;-----------------  cr0 第 0 位置 1  ----------------
44    mov eax, cr0
45    or eax, 0x00000001
46    mov cr0, eax
47
48    ; 刷新流水线，避免分支预测的影响，这种 CPU 优化策略，最怕 jmp 跳转，
```

```
49     ; 这将导致之前做的预测失效，从而起到了刷新的作用
50     jmp SELECTOR_CODE:p_mode_start
51
52 [bits 32]
53 p_mode_start:
54     mov ax, SELECTOR_DATA
55     mov ds, ax
56     mov es, ax
57     mov ss, ax
58     mov esp,0x900
59     push byte 0x7
60     push word 0x8
61     push dword 0x9
62     jmp $
```

代码似乎有点长，在第 46 行之前都是在为进入保护模式做准备，虽然我们目前还没有讲，但还是给大家呈上来了，先有个感性认识也好，实在没兴趣的话，请移步第 59 行，第 59～62 行同样是对三种操作数的入栈操作。表 4-5 是文件 32push.S 编译后上 CPU 的效果，表中信息同样是从 bochs 中摘录的重点。

表 4-5 保护模式下 push 指令操作数

行　　号	下一条指令	下一条的指令的机器码	当前 esp 值
1	push 0x00000007	6a07	0x00000900
2	push 0x0008	666a08	0x000008fc
3	push 0x00000009	6a09	0x000008fa
4	jmp .-2	ebfe	0x000008f6

同实模式的例子相比，表 4-5 中直接将更新 esp 值的步骤省了，这里也是将栈指针置为 0x900，还是为了方便大家检验栈指针的变化，不过这里是 32 位栈指针 esp。

在表 4-5 第 1 行中，下一条指令是 push 0x00000007，对应于文件 32push.S 的第 59 行 push byte 0x7（byte 是伪指令，表示 1 字节大小的数据类型，由编译器处理），通过对比大家看到，原本是压入 1 个字节的 0x7，现在变成了 4 个字节的 0x00000007，这同样也是压入 4 字节数据的节奏。果然，在第 2 行的当前 esp 值为 0x8fc，这是 0x900 减 4 的结果，也就是说栈指针 esp 减 4。

第 2 行的下一条指令是 push 0x0008，对应文件 32push.S 的第 60 行 push word 0x8（word 是伪指令，表示 2 字节大小的数据类型，由编译器处理）。其机器码为 666a08，其中低 1 字节是 0x66，这是操作数大小反转前缀。编译器添加此反转前缀的原因是在 32 位下的操作数是 4 字节，此处要压入 2 字节，这是 16 位模式下的操作数尺寸。到底是不是会压入 2 字节呢？让我们看看第三行的当前 esp 值，果然是 0x8fa，它是由上一次的栈指针 0x8fc 减 2 得来的。

第 3 行的指令对应文件 32push.S 的第 61 行 push dword 0x9（dword 是伪指令，表示 4 字节大小的数据类型，由编译器处理），这是本身是 32 位大小的操作数，所以直接入栈，栈指针应该减 4。到第 4 行的当前 esp 值验证，果然是 0x8f6，它是由 0x8fa 减 4 得到的。

对于段寄存器的入栈，即 cs、ds、es、fs、gs、ss，无论在哪种模式下，都是按当前模式的默认操作数大小压入的。例如，在 16 位模式下，CPU 直接压入 2 字节，栈指针 sp 减 2。在 32 位模式下，CPU 直接压入 4 字节，栈指针 esp 减 4。

好啦，你知道我总会用行动表示，直接上菜。

先看下实模式下压入段寄存器时 CPU 的表现，见代码 16sreg_push.S。

16sreg_push.S

```
1 section loader vstart=0x900
2 mov sp,  0x900
3 push cs
4 push ds
5 push es
6 jmp $
```

本程序在实模式下运行，闲话少说，直接上 CPU。

表 4-6　　实模式下段寄存器压栈

行号	下一条指令	下一条的指令的机器码	当前 esp 值
1	push cs	0x0e	0x900
2	push ds	0x1e	0x8fe
3	push es	0x06	0x8fc
4	jmp .-2	0xebfe	0x8fa

实模式下每次压入一个段寄存器，栈指针 sp 都会减 2。表 4-6 和之前的表 4-4、表 4-5 一样，大家自行验证吧。

再看下保护模式下 CPU 压入段寄存器的情况。

用于演示的代码文件 32sreg_push.S 大部分与 32push.S 一样，区别是第 59～61 行替换为以下三句。

```
......
59   push cs
60   push ds
61   push es
```

以上是 32sreg_push.S 的部分内容。真枪实弹上 CPU 后，见表 4-7。

表 4-7　　保护模式下段寄存器压栈

行号	下一条指令	下一条的指令的机器码	当前 esp 值
1	push cs	0x0e	0x900
2	push ds	0x1e	0x8fc
3	push es	0x06	0x8f8
4	jmp .-2	0xebfe	0x8f4

保护模式下每次压入一个段寄存器，栈指针 esp 都会减 4。大伙自行验证表 4-7 吧，哥们儿啥都不说了，无需解释您懂的。

对于通用寄存器和内存，无论是在实模式或保护模式：

- 如果压入的是 16 位数据，栈指针减 2。
- 如果压入的是 32 位数据，栈指针减 4。

咱们先验证下实模式压入 16 位、32 位数据后栈指针的变化，测试文件 16general_reg_mem_push.S。

16general_reg_mem_push.S

```
1 section loader vstart=0x900
2 mov sp, 0x900
3 push ax
4 push eax
5 push word [0x1234]
6 push dword [0x1234]
7 jmp $
```

本文件在第 3、4 行测试寄存器入栈，第 5、6 行是测试内存入栈。文件中第 2 行依然是将本指针 sp 指向 0x900，方便大伙儿查看。表 4-8 是上 CPU 后的真实情况。

表 4-8　　实模式下通用寄存器和内存压栈

行号	下一条指令	下一条的指令的机器码	当前 esp 值
1	push ax	50	0x900
2	push eax	6650	0x8fe
3	push word ptr ds：0x1234	ff363412	0x8fa
4	push dword ptr ds：0x1234	66ff363412	0x8f8
5	jmp .-2	ebfe	0x8f4

见表 4-8，第 1、2 行是压入通用寄存器。由于是在 16 位模式下测试，所以在第 2 行压入 32 位寄存器

时，编译器为指令添加了操作数大小反转前缀 0x66。

第 3、4 行是压入内存，第 3 行是压入 16 位内存数据，第 4 行是压入 32 位内存数据。由于当前模式是 16 位实模式，所以这两行的区别是编译器为第 4 行的指令添加了操作数大小反转前缀 0x66。

随着每次操作数的压入，栈指针 sp 的值每次都会减去操作数的大小，大伙儿根据表中数据自己验证下吧。

下面看看在保护模式下同样是压入通用寄存器和内存的情况。

用于演示的代码文件 32general_reg_mem_push.S 大部分与 32push.S 一样，区别是第 59～62 行替换为以下四行。

```
......
......
59   push ax
 60   push eax
 61   push word [0x1234]
 62   push dword [0x1234]
```

以上是 32general_reg_mem_push.S 部分内容。

表 4-9 是 32general_reg_mem_push.S 编译后在 CPU 上运行的真实情况。

表 4-9　保护模式下通用寄存器和内存压栈

行号	下一条指令	下一条的指令的机器码	当前 esp 值
1	push ax	6650	0x900
2	push eax	50	0x8fe
3	push word ptr ds：0x1234	66ff3534120000	0x8fa
4	push dword ptr ds：0x1234	ff3534120000	0x8f8
5	jmp .-2	ebfe	0x8f4

从表 4-9 中可以看出，在保护模式每次压入 16 位数据时栈指针 esp 就减 2，每次压入 32 位数据时栈指针 esp 就减 4。这里也在部分指令中添加了反转前缀 0x66。其他的内容不用我多说了，估计大家已经觉得我太啰嗦了，想必甚至觉得本段内容都是多余的，哈哈，大家理解了就好。

与保护模式的初次见面到此就结束了，真正能领略到保护模式风采的是后面的部分：全局描述符表、特权级、分页等，那些才是我们关注的重点。

4.3　全局描述符表

到了保护模式下，内存段（如数据段、代码段等）不再是简单地用段寄存器加载一下段基址就能用啦，段的信息增加了很多，需要提前把段定义好才能使用。就像家庭成员需要上户口一样，在户口簿上登记过才算合法。

全局描述符表（Global Descriptor Table，GDT）是保护模式下内存段的登记表，这是不同于实模式的显著特征之一。

先抛出这样一个术语来，下面会接着讲解。

4.3.1　段描述符

大家回想一下，首先，对于 IA32 架构的处理器（就是我们大多数人现在所用的处理器），访问内存采用“段基址：段内偏移地址”形式，即使到了保护模式，也是绕不开这个限制的，这是骨子里的问题。其次，为什么淘汰了实模式而发明了保护模式？最主要的是安全问题。基于以上两方面，CPU 工程师既要保证保护模式下的内存访问依然是“段基址：段内偏移”的形式，又要有效提高了安全性。

人所发明的东西都是基于人的思维设计的，CPU 的工程师们已经解决了这个问题。咱们也都是人，所以一定能够悟出工程师的做法。如果是您，您该怎样解决这个问题呢？之前在 16 位模式下，访问内存时只要将段基址加载到段寄存器中，再结合偏移地址就行了，段寄存器太小了，只能存储 16 位的信息，甚至连 20 位地

址都要借助左移 4 位来实现。现在为了安全性，总该为内存段添加一些额外的安全属性吧？问题来啦，这些用于安全方面的属性，该往哪放呢？寄存器就甭想了，即使是 32 位寄存器，也才刚刚够存放 32 位地址而已。排除了寄存器，自然只剩下内存可以考虑啦。

相对寄存器来说，内存可是非常大的，既然有了那么大的内存可用，那就干脆多添加一些信息，把安全做得更加彻底一些。

用哪些属性来描述这个内存段呢？

首先，先要解决实模式下存在的问题。

实模式下的用户程序可以破坏存储代码的内存区域，所以要添加个内存段类型属性来阻止这种行为。

实模式下的用户程序和操作系统是同一级别的，所以要添加个特权级属性来区分用户程序和操作系统的地位。

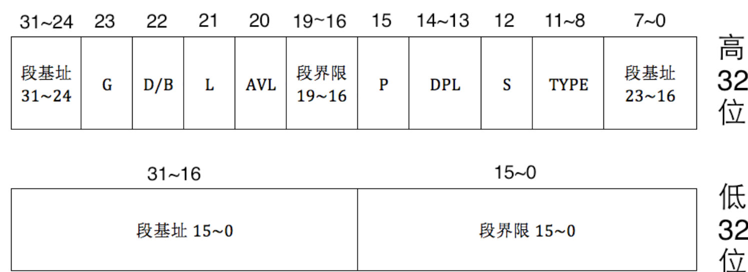
其次，是一些访问内存段的必要属性条件。

内存段是一片内存区域，访问内存就要提供段基址，所以要有段基址属性。

为了限制程序访问内存的范围，还要对段大小进行约束，所以要有段界限属性。

最后，要改进就改得彻底一些，所以多增加了一些约束条件，这些马上就会讲到。

您看，这里我只是说了一小部分内存段的属性，看上去也要占不少字节呢。这些用来描述内存段的属性，被放到了一个称为段描述符的结构中，顾名思义，该结构专门用来描述一个内存段，该结构是 8 字节大小（顺便说一句，在本书中提到的各种描述符大小都是 8 字节）。该结构如图 4-5 所示。



▲图 4-5　段描述符格式

跟大家说明一下，段描述符是 8 字节大小，在图 4-5 中我们为了方便展示，才将其“人为地”分成了低 32 位和高 32 位，即两个 4 字节部分。其实它们不能分成两部分，必须是连续的 8 字节，这样 CPU 才能读取到正确的段信息。

保护模式下地址总线宽度是 32 位，段基址需要用 32 位地址来表示。

段界限表示段边界的扩展最值，即最大扩展到多少或最小扩展到多少。扩展方向只有上下两种。对于数据段和代码段，段的扩展方向是向上，即地址越来越高，此时的段界限用来表示段内偏移的最大值。对于栈段，段的扩展方向是向下，即地址越来越低，此时的段界限用来表示段内偏移的最小值。无论是向上扩展，还是向下扩展，段界限的作用如同其名，表示段的边界、大小、范围。段界限用 20 个二进制位来表示。只不过此段界限只是个单位量，它的单位要么是字节，要么是 4KB，这是由描述符中的 G 位来指定的。最终段的边界是此段界限值*单位，故段的大小要么是 2 的 20 次方等于 1MB，要么是 2 的 32 次方（4KB 等于 2 的 12 次方，12+20=32）等于 4GB。

上面所说的 1MB 和 4GB 只是个范围，并不是具体的边界值。由于段界限只是个偏移量，是从 0 算起的，所以实际的段界限边界值=

（描述符中段界限+1）*（段界限的粒度大小：4KB 或者 1）-1。

这个公式很简单，就是表示有多少个 4KB 或 1。由于描述符中的段界限是从 0 起的，所以左边第 1 个括号中要加个 1，表示 4KB 或 1 的实际数量。它与第二个括号中的段粒度大小相乘后得到的乘积是以 1 为起始的段的实际大小。由于地址是以 0 为起始的，所以公式的最后又减了 1。

如果 G 位为 0，表示段界限粒度大小为 1 字节，根据上面的公式，实际段界限=（描述符中段界限+1）*1 -1=描述符中段界限，段界限实际大小就等于描述符中的段界限值。

如果 G 位为 1，表示段界限粒度大小为 4KB 字节，故实际段界限=（描述符中段界限+1）*4k-1。举个例子，如果是平坦模型，段界限为 0xFFFFF，G 位为 1，套用上面公式，段界限边界值=0x100000* 0x1000-1=0xFFFFFFFF。

内存访问需要用到“段其址：段内偏移地址”，段界限其实是用来限制段内偏移地址的，段内偏移地址必须位于段的范围之内，否则 CPU 会抛异常。根据段的扩展方向，此“段界限*单位”便是段内偏移地址的最大值（向上扩展）或最小值（向下扩展），任何超过此值的偏移地址都被认为是非法访问，CPU 会将此错误捕获。顺便提一句，是 CPU 硬件负责检测，这块没咱们啥事，但检测到错误后就有咱们的事啦，CPU 会触发相应的异常，咱们负责写相应的异常处理程序。

仔细看图 4-5，您会发现 20 位的段界限属性，居然被拆分成两部分。段界限的低 16 位（0～15 位）存放在段描述符的低 32 位，段界限的高 4 位（16～19 位）存放在段描述符的高 32 位。不止段界限这样，段基址更过分，32 位的段基址居然被分拆成三份存放。有没有觉得奇怪，这种格式确实太奇怪了，居然一个字段被拆分成多个……也许不用我说，很多读者都想到了，这属于历史遗留问题。为了兼容（主要是 Intel 公司的战略是把兼容放在第一位）CPU 不得不兼顾着过去的产品，这也正是计算机业能像今天这样朝气蓬勃发展的原因。

历史遗留问题？不知您心里是否也有个大大的问号。其实之前已经说过啦，80286 是第一款具有保护模式的 CPU，当时就已经采用段描述符来描述内存段信息了。只是当时它是 16 位的保护模式，地址总线是 24 位，最多访问 16MB 内存，产品定位模糊。所以这款产品只是用来试水的，最终也未像 8086 那样成功。但问题也就来了，虽然 80286 未有大的成功，但也依然有小部分市场（估计是卖给了当时的土豪们），把兼容性作为第一位的 Intel 不得不在原有 80286 的段描述符上做扩展，所以才有了今天这样奇形怪状的段描述符。

不过也不要太担忧这样会影响 CPU 获取段信息的效率，因为段信息会被 CPU 缓存到段描述符缓冲寄存器中，这个前面在介绍寄存器时有说过啦，此缓冲寄存器中的内容便是段描述符中的内容，它是经过 CPU 整理后的，段界限和段基址已经被拼合到一起啦，CPU 下次会自动到段描述符缓冲寄存器中取段数据。

下面根据图 4-5，介绍一下段描述符中的属性。为了方便大家阅读，对字段的解释将以段描述符中的字段顺序为主，也许内容会显得有些虎头蛇尾，大伙儿多多包涵。

咱们先从段描述符的低 32 位开始。

段描述符的低 32 位分为两部分，前 16 位用来存储段的段界限的前 0～15 位，后 16 位用来存储段基址的 0～15 位。

主要的属性都在段描述符的高 32 位，一眼望去可不少呢，现在我开始逐一和大家介绍。注意，以下内容说的都是在段描述符的高 32 位中，为叙述方便不再重复说明。

0～7 位是段基址的 16～23，24～31 位是段基址的 24～31 位，加上在段描述符低 32 位中的段基址 0～15 位，这下 32 位基地址才算齐全了。

8～11 位是 type 字段，共 4 位，用来指定本描述符的类型。这里要提前说下段描述符的 S 字段了。是这样的，一个段描述符，在 CPU 眼里分为两大类，要么描述的是系统段，要么描述的是数据段，这是由段描述符中的 S 位决定的，用它指示是否是系统段。在 CPU 眼里，凡是硬件运行需要用到的东西都可称之为系统，凡是软件（操作系统也属于软件，CPU 眼中，它与用户程序无区别）需要的东西都称为数据，无论是代码，还是数据，甚至包括栈，它们都作为硬件的输入，都是给硬件的数据而已，所以代码段在段描述符中也属于数据段（非系统段）。S 为 0 时表示系统段，S 为 1 时表示数据段。type 字段是要和 S 字段配合在一起才能确定段描述符的确切类型，只有 S 字段的值确定后，type 字段的值才有具体意义。

什么是系统段？各种称为“门”的结构便是系统段，也就是硬件系统需要的结构，非软件使用的，如调用门、任务门。简而言之，门的意思就是入口，它通往一段程序。关于系统段这里咱们不再多说，目前主要是关注 S 为 1 时，非系统段的 type 子类型。

赶紧回来继续说 type 字段，该字段共 4 位，用于表示内存段或门的子类型。说明见表 4-10。

表 4-10 段描述符的 type 类型

	系统段类型	第 3～0 位				说明
		3	2	1	0	
系统段	未定义	0	0	0	0	保留
	可用的 80286 TSS	0	0	0	1	仅限 286 的任务状态段
	LDT	0	0	1	0	局部描述符表，只有第 1 位为 1
	忙碌的 80286 TSS	0	0	1	1	仅限 286。type 中的第 1 位称为 B 位，若为 1，则表示当前任务忙碌。由 CPU 将此位置 1
	80286 调用门	0	1	0	0	仅限 286
	任务门	0	1	0	1	任务门在现代操作系统中很少用到
	80286 中断门	0	1	1	0	仅限 286
	80286 陷阱门	0	1	1	1	仅限 286
	未定义	1	0	0	0	保留
	可用的 80386TSS	1	0	0	1	386 以上 CPU 的 TSS，type 第 3 位为 1
	未定义	1	0	1	0	保留
	忙碌的 80386 TSS	1	0	1	1	386 以上 CPU 的 TSS，type 第 3 位为 1
	80386 调用门	1	1	0	0	386 以上 CPU 的调用门，type 第 3 位为 1
	未定义	1	1	0	1	保留
	中断门	1	1	1	0	386 以上 CPU 的中断门
	陷阱门	1	1	1	1	386 以上 CPU 的陷阱门
对于非系统段，按代码段和数据段划分，这 4 位分别有不同的意义						
非系统段	内存段类型	X	R	C	A	说明
	代码段	1	0	0	*	只执行代码段
		1	1	0	*	可执行、可读代码段
		1	0	1	*	可执行、一致性代码段
		1	1	1	*	可执行、可读、一致性代码段
	内存段类型	X	W	E	A	说明
	数据段	0	0	0	*	只读数据段
		0	1	0	*	可读写数据段
		0	0	1	*	只读，向下扩展的数据段
		0	1	1	*	可读写，向下扩展的数据段

虽然表中一并列出了系统段，但咱们目前还是主要关注非系统段，部分非系统段会在今后的内容中讲述。

表中的 A 位表示 Accessed 位，这是由 CPU 来设置的，每当该段被 CPU 访问过后，CPU 就将此位置 1。所以，创建一个新段描述符时，应该将此位置 0。我们在调试时，根据此位便能判断该描述符是否可用啦。

C 表示一致性代码段，也称为依从代码段，Conforming。一致性代码段是指如果自己是转移的目标段，并且自己是一致性代码段，自己的特权级一定要高于当前特权级，转移后的特权级不与自己的 DPL 为主，而是与转移前的低特权级一致，也就是听从、依从转移前的低特权级。C 为 1 时则表示该段是一致性代码段，C 为 0 时则表示该段为非一致性代码段。

R 表示可读，R 为 1 表示可读，R 为 0 表示不可读。这个属性一般用来限制代码段的访问。如果指令执行过程中，CPU 发现某些指令对 R 为 0 的段进行访问，如使用段超越前缀 CS 来访问代码段，CPU 将抛出异常。啰嗦一小下，内存中的数据对 CPU 来说是要处理的数据，仅仅是 CPU 的输入而已，CPU 的铁骑可以踏遍任意角落。所以，不可读的代码段只是来限制代码指令的，并不是连 CPU 也不能看。

X 表示该段是否可执行，EXecutable。我们所说的指令和数据，在 CPU 眼中是没有任何区别的，都是 010101 这样类似的二进制。所以要用 type 中的 X 位来标识出是否是可执行的代码。代码段是可执行的，即 X 为 1。而数据段是不可执行的，即 X 为 0。

E 是用来标识段的扩展方向，Extend。E 为 0 表示向上扩展，即地址越来越高，通常用于代码段和数据段。E 为 1 表示向下扩展，地址越来越低，通常用于栈段。

W 是指段是否可写，Writable。W 为 1 表示可写，通常用于数据段。W 为 0

表示不可写入，通常用于代码段。对于 W 为 0 的段有写入行为，同样会引发 CPU 抛出异常。

段描述符的第 12 位是 S 字段，前面在介绍 type 时已解释过啦，用来指出当前描述符是否是系统段。S 为 0 表示系统段，S 为 1 表示非系统段。关于系统段的类型，可参见表 4-10 的系统段类型部分。

段描述符的第 13～14 位是 DPL 字段，Descriptor Privilege Level，即描述符特权级，这是保护模式提供的安全解决方案，将计算机世界按权力划分成不同等级，每一种等级称为一种特权级。

由于段描述符用来描述一个内存段或一段代码的情况（若描述符类型为“门”），所以描述符中的 DPL 是指所代表的内存段的特权级。

这两位能表示 4 种特权级，分别是 0、1、2、3 级特权，数字越小，特权级越大。特权级是保护模式下才有的东西，CPU 由实模式进入保护模式后，特权级自动为 0。因为保护模式下的代码已经是操作系统的一部分啦，所以操作系统应该处于最高的 0 特权级。用户程序通常处于 3 特权级，权限最小。某些指令只能在 0 特权级下执行，从而保证了安全。

段描述符的第 15 位是 P 字段，Present，即段是否存在。如果段存在于内存中，P 为 1，否则 P 为 0。P 字段是由 CPU 来检查的，如果为 0，CPU 将抛出异常，转到相应的异常处理程序，此异常处理程序是咱们来写的，在异常处理程序处理完成后要将 P 置 1。也就是说，对于 P 字段，CPU 只负责检查，咱们负责赋值。不过在通常情况下，段都是在内存中的。当初 CPU 的设计是当内存不足时，可以将段描述符中对应的内存段换出，也就是可以把不常用的段直接换出到硬盘，待使用时再加载进来。但现在即使内存不足时，也没有将整个段都换出去的，现在基本都是平坦模型，一般情况下，段都要 4GB 大小，换到硬盘不也是很占空间吗？而且这些平坦的段都是公用的，换出去就麻烦啦。所以这些是未开启分页时的解决方案，保护模式下有分页功能，可以按页（4KB）的单位来将内存换入换出。

段描述符的第 16～19 位是段界限的第 16～19 位。这样共 20 位的段界限就齐全啦。

段描述符的第 20 位为 AVL 字段，从名字上看它是 AVaiLable，可用的。不过这“可用的”是对用户来说的，也就是操作系统可以随意用此位。对硬件来说，它没有专门的用途，就当作是硬件给软件的馈赠吧。

段描述符的第 21 位为 L 字段，用来设置是否是 64 位代码段。L 为 1 表示 64 位代码段，否则表示 32 位代码段。这目前属于保留位，在我们 32 位 CPU 下编程，将其置为 0 便可。

段描述符的第 22 位是 D/B 字段，用来指示有效地址（段内偏移地址）及操作数的大小。有没有觉得奇怪，实模式已经是 32 位的地址线和操作数了，难道操作数不是 32 位大小吗？其实这是为了兼容 286 的保护模式，286 的保护模式下的操作数是 16 位。既然是指定“操作数”的大小，也就是对“指令”来说的，与指令相关的内存段是代码段和栈段，所以此字段是 D 或 B。

对于代码段来说，此位是 D 位，若 D 为 0，表示指令中的有效地址和操作数是 16 位，指令有效地址用 IP 寄存器。若 D 为 1，表示指令中的有效地址及操作数是 32 位，指令有效地址用 EIP 寄存器。

对于栈段来说，此位是 B 位，用来指定操作数大小，此操作数涉及到栈指针寄存器的选择及栈的地址上限。若 B 为 0，使用的是 sp 寄存器，也就是栈的起始地址是 16 位寄存器的最大寻址范围，0xFFFF。若 B 为 1，使用的是 esp 寄存器，也就是栈的起始地址是 32 位寄存器的最大寻址范围，0xFFFFFFFF。

段描述符的第 23 位是 G 字段，Granularity，粒度，用来指定段界限的单位大小。所以此位是用来配合段界限的，它与段界限一起来决定段的大小。若 G 为 0，表示段界限的单位是 1 字节，这样段最大是 2 的 20 次方*1 字节，即 1MB。若 G 为 1，表示段界限的单位是 4KB，这样段最大是 2 的 20 次方*4KB 字节，即 4GB。

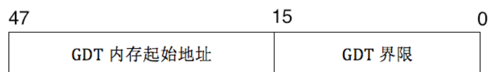
段描述符的第 24～31 位是段基址的第 24～31 位，这是段基址的最后 8 位。

把段描述符说完了，其实以后还要说其他描述符呢，所以，用不着把每个字段的意义都记下来，将来用的时候再回来翻看才省事方便。

4.3.2　全局描述符表 GDT、局部描述符表 LDT 及选择子

一个段描述符只用来定义（描述）一个内存段。代码段要占用一个段描述符、数据段和栈段等，多个内存段也要各自占用一个段描述符，这些描述符放在哪里呢？答案是放在全局描述符表，就是本节开头所说的 GDT（Global Descriptor Table）。全局描述符表 GDT 相当于是描述符的数组，数组中的每个元素都是 8 字节的描述符。可以用选择子（马上会讲到）中提供的下标在 GDT 中索引描述符。

为什么将该表称为“全局”描述符表？全局体现在多个程序都可以在里面定义自己的段描述符，是公用的。全局描述符表位于内存中，需要用专门的寄存器指向它后，CPU 才知道它在哪里。这个专门的寄存器便是 GDTR，即 GDT Register，专门用来存储 GDT 的内存地址及大小。GDTR 是个 48 位的寄存器，如图 4-6 所示。



▲图 4-6　GDTR 寄存器

不过，对此寄存器的访问，不能够用 mov gdtr，xxx 这样的指令为 gdtr 初始化，有专门的指令来做这件事，这就是 lgdt 指令。虽然我们是为了进入保护模式才讲述的 lgdt，因此看上去此指令是在实模式下执行的，但实际上，此指令在保护模式下也能够执行。言外之意便是进入保护模式需要有 GDT，但进入保护模式后，还可以再重新换个 GDT 加载。在保护模式下重新换个 GDT 的原因是实模式下只能访问低端 1MB 空间，所以 GDT 只能位于 1MB 之内。根据操作系统的实际情况，有可能需要把 GDT 放在其他的内存位置，所以在进入保护模式后，访问的内存空间突破了 1MB，可以将 GDT 放在合适的位置后再重新加载进来。

lgdt 的指令格式是：lgdt48 位内存数据。

这 48 位内存数据划分为两部分，其中前 16 位是 GDT 以字节为单位的界限值，所以这 16 位相当于 GDT 的字节大小减 1。后 32 位是 GDT 的起始地址。由于 GDT 的大小是 16 位二进制，其表示的范围是 2 的 16 次方等于 65536 字节。每个描述符大小是 8 字节，故，GDT 中最多可容纳的描述符数量是 65536/8=8192 个，即 GDT 中可容纳 8192 个段或门。

保护模式中的段描述符虽然看上去又怪异又复杂，但它本质上只是一段内存区域的“身份证”而已，尽管它不像实模式那样直接，即段的大小统一都是 64KB，段基址代表内存的起始，偏移地址代表段内偏移量……但它代表的同样也是一段内存。段描述符与内存段的关系如图 4-7 所示。

段描述符有了，描述符表也有了，我们该如何使用它呢？下面我们引出新的概念：段的选择子。

段寄存器 CS、DS、ES、FS、GS、SS，在实模式下时，段中存储的是段基地址，即内存段的起始地址。而在保护模式下时，由于段基址已经存入了段描述符中，所以段寄存器中再存放段基址是没有意义的，在段寄存器中存入的是一个叫作选择子的东西——selector。选择子“基本上”是个索引值，这里说的是基本上，其中还有其他属性咱们一会再说。用此索引值在段描述符表中索引相应的段描述符，这样，便在段描述符中得到了内存段的起始地址和段界限值等相关信息。

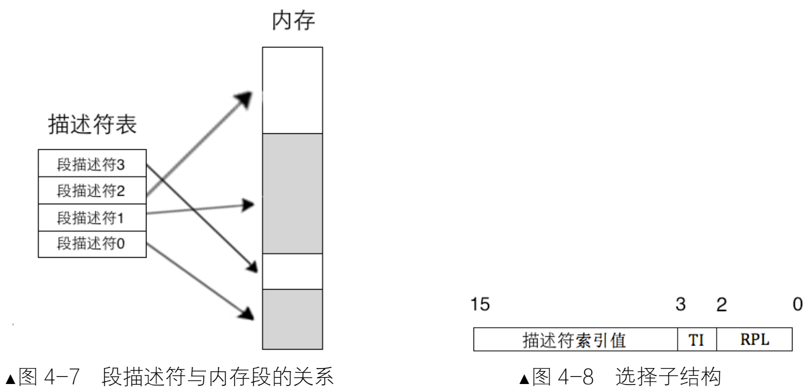
下面正式介绍下选择子。由于段寄存器是 16 位，所以选择子也是 16 位，在其低 2 位即第 0～1 位，用来存储 RPL，即请求特权级，可以表示 0、1、2、3 四种特权级。关于 RPL 我们会在专门讲特权级的章节中详尽说明，此处可以理解为请求者的当前特权级（不理解也没关系，因为在本章中它不重要）。在选择子的第 2 位是 TI 位，即 Table Indicator，用来指示选择子是在 GDT 中，还是 LDT 中索引描述符。TI 为 0 表示在 GDT 中索引描述符，TI 为 1 表示在 LDT 中索引描述符。选择子的高 13 位，即第 3～15 位是描述符的索引值，用此值在 GDT 中索引描述符。前面说过 GDT 相当于一个描述符数组，所以此选择子中的索引值就是 GDT 中的下标。选择子结构如图 4-8 所示。

由于选择子的索引值部分是 13 位，即 2 的 13 次方是 8192，故最多可以索引 8192 个段，这和 GDT 中最多定义 8192 个描述符是吻合的。

选择子的作用主要是确定段描述符，确定描述符的目的，一是为了特权级、界限等安全考虑，最主要的还是要确定段的基地址。

虽然到了保护模式，但 IA32 架构始终脱离不了内存分段，即访问内存必须要用“段基址：段内偏移地址”的形式。保护模式下的段寄存器中已经是选择子，不再是直接的段基址。段基址在段描述符中，用

给出的选择子索引到描述符后，CPU 自动从段描述符中取出段基址，这样再加上段内偏移地址，便凑成了“段基址：段内偏移地址”的形式。



▲图 4-7　段描述符与内存段的关系　　▲图 4-8　选择子结构

但大家要注意，到了保护模式下后，由于已经是 32 位地址线和 32 位寄存器啦，任意一寄存器都能够提供 32 位地址，故不需要再将段基址乘以 16 后再与段内偏移地址相加啦，直接用选择子对应的“段描述符中的段基址”加上“段内偏移地址”就是要访问的内存地址。

例如选择子是 0x8，将其加载到 ds 寄存器后，访问 ds：0x9 这样的内存，其过程是：0x8 的低 2 位是 RPL，其值为 00。第 2 是 TI，其值 0，表示是在 GDT 中索引段描述符。用 0x8 的高 13 位 0x1 在 GDT 中索引，也就是 GDT 中的第 1 个段描述符（GDT 中第 0 个段描述符不可用）。假设第 1 个段描述符中的 3 个段基址部分，其值为 0x1234。CPU 将 0x1234 作为段基址，与段内偏移地址 0x9 相加，0x1234+0x9=0x123d。用所得的和 0x123d 作为访存地址。

值得注意的是上面括号中提到了 GDT 中的第 0 个段描述符是不可用的，原因是定义在 GDT 中的段描述符是要用选择子来访问的，如果使用的选择子忘记初始化，选择子的值便会是 0，这便会访问到第 0 个段描述符。为了避免出现这种因忘记初始化选择子而选择到第 0 个段描述符的情况，GDT 中的第 0 个段描述符不可用。也就是说，若选择到了 GDT 中的第 0 个描述符，处理器将发出异常。

按理说有全局就要有局部，还真有，这就是局部描述符表，叫 LDT，Local Descriptor Table，它是 CPU 厂商为在硬件一级原生支持多任务而创造的表，按照 CPU 的设想，一个任务对应一个 LDT。其实在现代操作系统中很少有用 LDT 的，我们系统中也未用 LDT。所以这里就捎带着说一下，点到为止。

CPU 厂商建议每个任务的私有内存段都应该放到自己的段描述符表中，该表就是 LDT，即每个任务都有自己的 LDT，随着任务切换，也要切换相应任务的 LDT。LDT 也位于内存中，其地址需要先被加载到某个寄存器后，CPU 才能使用 LDT，该寄存器是 LDTR，即 LDT Register。同样也有专门的指令用于加载 LDT，即 lldt。以后每切换任务时，都要用 lldt 指令重新加载任务的私有内存段。

回顾一下段描述符中的 type 字段，其中 LDT 为系统段，换句话说，LDT 虽然是个表，但其也是一片内存区域，所以也需要用个描述符在 GDT 中先注册。段描述符是需要用选择子去访问的。故，lldt 的指令格式为：

lldt 16 位寄存器/16 位内存

无论是寄存器，还是内存，其内容一定是个选择子，该选择子用来在 GDT 中索引 LDT 的段描述符。

在 LDT 被加载到 ldtr 寄存器后，之后再访问某个段时，选择子中的 TI 位若为 1，就会用该选择子中的高 13 位在 ldtr 寄存器所指向的 LDT 中去索引相应段描述符。

LDT 中的段描述符和 GDT 中的一样，与 GDT 不同的是 LDT 中的第 0 个段描述符是可用的，因为提交的选择子中的 TI 位，TI 位用于指定是 GDT，还是 LDT，TI 为 1 则表示在 LDT 中索引段描述符，即 TI 为 1 必然是经过显式初始化的结果，完全排除了忘记初始化的可能。好啦，LDT 就说到这，以后在介绍 TSS 时还会说一些 LDT 的内容。

在表 4-10 中列出的系统段，它们都有各自的描述符，或者说描述符结构相同，只是用 type 和 S 字段予以区分不同的描述符。任何描述符的大小都是 8 字节。但无论描述符的种类有多少，它们的高 32 位中的第 8～12 字

节内容都是不变的，第 12 位必须是 S，第 8～11 位必须为 type，描述符中的其他位的内容不强制要求。这样“局部”格式统一，便于 CPU 识别段类型。虽然列出了众多系统段，但很多都不是在全局描述符表中的，如中断门、陷阱门、任务门等都是在中断描述符表中存在的，将来说到中断的时候咱们再展开这方面内容。

4.3.3 打开 A20 地址线

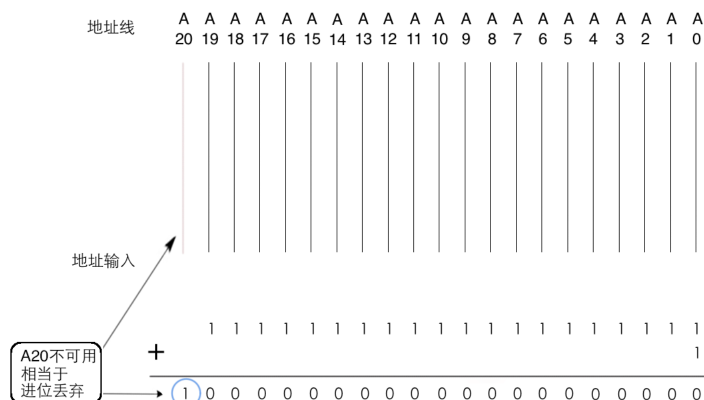
还记得实模式下的 wrap-around 吗？也就是地址回绕。咱们一起来复习一下。

实模式下内存访问是采取“段基址：段内偏移地址”的形式，段基址要乘以 16 后再加上段内偏移地址。实模式下寄存器都是 16 位的，如果段基址和段内偏移地址都为 16 位的最大值，即 0xFFFF：0xFFFF，最大地址是 0xFFFF0+0xFFFF，即 0x10FFEF。由于实模式下的地址线是 20 位，最大寻址空间是 1MB，即 0x00000～0xFFFFF。超出 1MB 内存的部分在逻辑上也是正常的，但物理内存中却没有与之对应的部分。为了让“段基址：段内偏移地址”策略继续可用，CPU 采取的做法是将超过 1MB 的部分自动回绕到 0 地址，继续从 0 地址开始映射。相当于把地址对 1MB 求模。超过 1MB 多余出来的内存被称为高端内存区 HMA。

这种地址回绕是如何做到的呢？要分两种情况分别讨论啦。

对于只有 20 位地址线的 CPU，不需要任何额外操作便能自动实现地址回绕。

地址（Address）线从 0 开始编号，在 8086/8088 中，只有 20 位地址线，即 A0～A19。20 位地址线表示的内存是 2 的 20 次方，最大是 1MB，即 0x0～0xFFFFF。内存若超过 1MB，是需要第 21 条地址线支持的。所以说，若地址进位到 1MB 以上，如 0x100000，由于没有第 21 位地址线，相当于丢掉了进位 1，变成了 0x00000。这一“缺陷”甚至成了当时很多程序员利用的技巧。地址回绕如图 4-9 所示。



▲图 4-9 地址回绕

对于 80286 后续的 CPU，通过 A20GATE 来控制 A20 地址线。

CPU 发展到了 80286 后，虽然地址总线从原来的 20 位发展到了 24 位，从而能够访问的内存范围可达到 2 的 24 次方，等于 16MB。但任何时候，Intel 都会把兼容放在第一位。80286 是第一款具有保护模式的 CPU，它在实模式下时，其表现也应该和 8086/8088 一模一样。按照兼容的要求，这意味着 80286 以及后续 CPU 的实模式都应该与 8086/8088 完全一样，即仍然只使用 20 条地址线。但 80286 有 24 条地址线，即 A0～A23，也就是说 A20 地址线是开启的。如果访问 0x100000～0x10FFEF 之间的内存，系统将直接访问这块物理内存，并不会像 8086/8088 那样回绕到 0。

为了解决此问题，IBM 在键盘控制器上的一些输出线来控制第 21 根地址线（A20）的有效性，故被称为 A20Gate。

- 如果 A20Gate 被打开，当访问到 0x100000～0x10FFEF 之间的地址时，CPU 将真正访问这块物理内存。
- 如果 A20Gate 被禁止，当访问 0x100000～0x10FFEF 之间的地址时，CPU 将采用 8086/8088 的地址回绕。

上面描述了地址回绕的原理，但地址回绕是为了兼容 8086/8088 的实模式。如今我们是在保护模式下，我们需要突破第 20 条地址线（A20）去访问更大的内存空间。而这一切，只有关闭了地址回绕才能实现。而关闭地址回绕，就是上面所说的打开 A20Gate。

说了半天，其实打开 A20Gate 的方式是极其简单的，将端口 0x92 的第 1 位置 1 就可以了，以下三个步骤就可以实现啦。

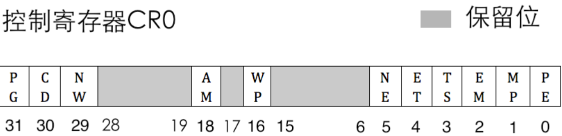
```
in al，0x92
or al，0000_0010B
out 0x92，al
```

我们离保护模式越来越近了，为了给大家一个盼头，剧透一下，其实我们距 32 位地址空间只差一步，下一节中我们将播出有关内容，欢迎大家准时收看。

4.3.4　保护模式的开关，CR0 寄存器的 PE 位

这是进入模式的最后一步，也就是第三步。这一步过后，我们将突破 1MB 内存的束缚，踏入广阔 4G 的天空，准备大干一场啦。

在前面说寄存器的时候，有讲到过控制寄存器系列 CRx。控制寄存器是 CPU 的窗口，既可以用来展示 CPU 的内部状态，也可用于控制 CPU 的运行机制。这次我们要用到的是 CR0 寄存器。更准确地说，我们要用到 CR0 寄存器的第 0 位，即 PE 位，Protection Enable，此位用于启用保护模式，是保护模式的开关，对保护模式来说，有万事俱备只欠东风的感觉。当打开此位后，CPU 才真正进入保护模式，所以这是进入保护模式三步中的最后一步。图 4-10 所示是 32 位 CR0 全貌。



▲图 4-10　控制寄存器 CR0

虽然都说啦目前只需要关注 PE 位，但考虑到一些和我一样有强迫症的同学，我还是把 CR0 中的字段全列出来啦，见表 4-11，描述部分没有换成对应的中文，老外发明的东西还是用原汁原味的英文好。

表 4-11　　控制寄存器 CR0 字段

标　志　位	描　　述
PE	Protection Enable
MP	Monitor coProcessor/Math Present
EM	Emulation
TS	Task Switched
ET	Extention Type
NE	Numeric Error
WP	Write Protect
AM	Alignment Mask
NW	Not Writethrough
CD	Cache Disable
PG	Paging

PE 为 0 表示在实模式下运行，PE 为 1 表示在保护模式下运行。所以，我们的任务是将此位置 1。示例代码如下。

```
1 mov eax, cr0
2 or eax, 0x00000001
3 mov cr0, eax
```

第 1 行代码是将 cr0 写入 eax。

第 2 行代码，通过或运算 or 指令将 eax 的第 0 位置 1。

第 3 行是将 eax 写回 cr0，这样 cr0 的 PE 位便为 1 了。

好啦，进入保护模式的条件咱们都说完啦。好久都没碰代码啦，是时候进入保护模式大干一场啦。

4.3.5　让我们进入保护模式

如今我们已经知道如何进入保护模式了，下面咱们通过实践，把以上知识贯穿起来。

其实在前面讲述保护模式下的指令扩展时，就已经“见识”了保护模式，就是那个文件 32push.S。不过此文件的保护模式并不直观，我们对它稍微改造一下，让其变得“可见”……我就不故弄玄虚了，其实就是打印一些字符，哈哈。

上正餐之前，还是先给大家来点小甜点，有些相关的东东要提前和大家交待清楚。

保护模式是在 loader.bin 中进入的，除了源程序 loader.S 要更新外，还更新了相关的 2 个文件。

第一个是 mbr.S，由于 loader.bin 超过了 512 字节，所以我们要把 mbr.S 中加载 loader.bin 的读入扇区数增大，目前它是 1 扇区，为了避免将来再次修改，直接改成读入 4 扇区。见代码 4-1 的 52 行。

代码 4-1 （project/c4/a/boot/mbr.S）

```
…略
52    mov cx,4              ; 待读入的扇区数
53    call rd_disk_m_16     ; 以下读取程序的起始部分（一个扇区）
…略
```

loader.bin 是由 mbr.bin 中的函数 rd_disk_m_16 负责加载的，其参数“读入扇区数”存入 cx 寄存器中。所以，如果 loader.bin 的大小超过 mbr 所读取的扇区数，切记一定要修改 mbr.S 中函数 rd_disk_m_16 的读入扇区数，如代码 4-2 中的第 52 行。如果忘记的话，由于被加载的程序不全，CPU 在执行时就会执行到一些莫名其妙的代码（内存中的数据“恰好”符合某种指令，虽然用了“恰好”，但这种情况很普遍，一堆 01 二进制串，总能瞎猫碰上死耗子成为某个指令），查看反汇编代码，根本就不是自己写的指令。我经常因为忘记修改这个参数而瞎折腾好久，每次找到原因时都是热泪盈眶，甚至满地打滚儿……哈哈，您懂的，当程序调通时，很多程序员都是喜大普奔的样子。

另一个要更新的文件是 include/boot.inc，里面是一些配置信息，loader.S 中用到的配置都是定义在 boot.inc 中的符号，见代码 4-2。

代码 4-2 （project/c4/a/boot/include/boot.inc）

```
  1 ;-------------   loader 和 kernel   ----------
  2
  3 LOADER_BASE_ADDR equ 0x900
  4 LOADER_START_SECTOR equ 0x2
  5
  6 ;--------------   gdt 描述符属性  -------------
  7 DESC_G_4K   equ   1_00000000000000000000000b
  8 DESC_D_32   equ    1_0000000000000000000000b
  9 DESC_L      equ     0_000000000000000000000b
 ; 64 位代码标记,此处标记为 0 便可
 10 DESC_AVL    equ      0_00000000000000000000b
                 ; CPU 不用此位,暂置为 0
 11 DESC_LIMIT_CODE2  equ 1111_0000000000000000b
 12 DESC_LIMIT_DATA2  equ DESC_LIMIT_CODE2
 13 DESC_LIMIT_VIDEO2 equ 0000_000000000000000b
 14 DESC_P      equ      1_000000000000000b
 15 DESC_DPL_0  equ     00_0000000000000b
 16 DESC_DPL_1  equ    01_0000000000000b
 17 DESC_DPL_2  equ       10_0000000000000b
 18 DESC_DPL_3  equ       11_0000000000000b
 19 DESC_S_CODE equ         1_000000000000b
 20 DESC_S_DATA equ   DESC_S_CODE
 21 DESC_S_sys  equ      0_000000000000b
 22 DESC_TYPE_CODE  equ         1000_00000000b
 ;x=1,c=0,r=0,a=0 代码段是可执行的，非一致性，不可读，已访问位 a 清 0

 23DESC_TYPE_DATA  equ         0010_00000000b
 ;x=0,e=0,w=1,a=0 数据段是不可执行的，向上扩展的，可写，已访问位 a 清 0
 24
 25 DESC_CODE_HIGH4 equ (0x00 << 24) + DESC_G_4K + DESC_D_32 + \
 DESC_L + DESC_AVL + DESC_LIMIT_CODE2 + \
 DESC_P+DESC_DPL_0 + DESC_S_CODE +\
 DESC_TYPE_CODE + 0x00

 26 DESC_DATA_HIGH4 equ (0x00 << 24) + DESC_G_4K + DESC_D_32 +\
 DESC_L + DESC_AVL + DESC_LIMIT_DATA2 + \
```

```
DESC_P + DESC_DPL_0 + DESC_S_DATA + \
DESC_TYPE_DATA + 0x00

27 DESC_VIDEO_HIGH4 equ (0x00 << 24) + DESC_G_4K + DESC_D_32 +\
DESC_L + DESC_AVL + DESC_LIMIT_VIDEO2 + DESC_P + \
DESC_DPL_0 + DESC_S_DATA + DESC_TYPE_D ATA + 0x00
28
29 ;--------------   选择子属性  ---------------
30 RPL0  equ   00b
31 RPL1  equ   01b
32 RPL2  equ   10b
33 RPL3  equ   11b
34 TI_GDT   equ   000b
35 TI_LDT   equ   100b
```

代码 4-2 中，主要是新增段描述符的属性及选择子，都是以宏的方式实现的。

equ 是 nasm 提供的伪指令，意为 equal，即等于，用于给表达式起个意义更明确的符号名，其指令格式是：

符号名称 equ 表达式

描述符中的各个字段都是由 equ 来定义的，符号名一律采用“DESC_字段名_字段相关信息”的形式。例如符号 DESC_G_4K，这就表示描述符的 G 位为 4K 粒度，其值等于（equ）1_00000000000000000000000b。其中结尾的 b 表示二进制，之所以这样用二进制写属性位，就是为了在格式中的位置容易对比，最左边的 1 正好处在第 23 位，也就是段描述符中 G 的位次。1 右边的字符“_”没有特别的意义，只是我人为加上去的，这样人眼“看”起来显得比较清晰明朗，nasm 编译器做得很人性化，为了人看得方便，它特意支持这种分隔符的写法，在编译阶段会忽略此分隔符。也许 type 字段的定义更有说服力，见第 22 行：“DESC_TYPE_CODE　equ　1000_00000000b”，这是定义了一个代码段的 type 字段，右边的二进制串的高 4 位就是 type 中的 4 位，右边用'_'字符来分隔，确实直观了很多，如果您忘记了 type 中各位的意义，赶紧回去翻看段描述符格式。此定义的意义在注释中已写得很清楚了：x=1，c=0，r=0，a=0 代码段是可执行的，非一致性，不可读，已访问位 a 清 0。

在保护模式中，我们还是学习 Linux 等主流操作系统的内存段，用平坦模型。平坦模型之前已经提到过了，就是整个内存都在一个段里，不用再像实模式那样用切换段基址的方式访问整个地址空间。在 32 位保护模式中，寻址空间是 4G，所以，平坦模型在我们定义的描述符中，段基址是 0，段界限*粒度等于 4G。粒度我们选的是 4k，故段界限是 0xFFFFF。下面以第 25 行的段定义为例。

第 25 行的 DESC_CODE_HIGH4，就是定义了代码段的高 4 字节。equ 后面那一串加法表达式，就是在凑足段描述符这高 4 字节内容。其中（0x00 << 24）表示“段基址 24～31”字段，该字段位于段描述符高 4 字节中的第 24～31 位，由于平坦模式段基址是 0，所以咱们用 0 偏移 24 位填充该字段。当然这只是一部分段基址，段基址在 8 字节的段描述符中存在 3 处，它们在每处都会是 0。继续看，DESC_G_4K 表示 4k 的粒度，其定义刚才说过啦。DESC_D_32 表示描述符中的 D/B 字段，对代码段来说是 D 位，在此表示 32 位操作数。DESC_L 表示段描述符中的 L 位，其值见代码 4-2 的第 9 行，为 0，表示为 32 位代码段。DESC_AVL 为 0，前面介绍过啦，此位没实际意义，是留给操作系统用的。DESC_LIMIT_CODE2 是代码段的段界限的第 2 部分（段界限的第 1 部分在段描述符的低 4 字节中），此处值为 1111b，它与段界限的第 1 部分将组成 20 个二进制 1，即总共的段界限将是 0xFFFFF。DESC_P 表示段存在。DESC_DPL_0 表示该段描述符对应的内存段的特权级是 0，即最高特权级。当 CPU 在该段上运行时，将有至高无上的特权。DESC_S_CODE 是代码段的 S 位，此值为 1，表示它是个普通的内存段，不是系统段。DESC_TYPE_CODE 刚才刚说过，意义为 x=1，c=0，r=0，a=0 ，即代码段是可执行的，非一致性，不可读，已访问位 a 清 0。0x00 是段基址的第 16～23 位，位于段描述符高 4 字节的起始 8 位，如前所述，由于是平坦模型，所以段基址的任意部分都是 0。

第 29～35 行是定义选择子属性，字面上很好理解，不多说了。这里并没有把选择子定义到这里，因为选择子中的高 13 位是用来索引段描述符的，它的值取决于段描述符的具体位置，而段描述符我们在 loader.S 中定义，所以最终的选择子是在 loader.S 中定义的，这样修改段描述符的位置时，顺便就把选择子修改了，否则放在多个文件中容易遗漏。

介绍完了相关文件，这下可以上正餐啦，下面是 loader.S 文件，让我们看看它是怎么进入保护模式的，

上代码。

代码 4-3 （project/c4/a/boot/loader.S）

```
1   %include "boot.inc"
2   section loader vstart=LOADER_BASE_ADDR
3   LOADER_STACK_TOP equ LOADER_BASE_ADDR
4   jmp loader_start
5
6 ;构建 gdt 及其内部的描述符
7   GDT_BASE:  dd   0x00000000
8               dd   0x00000000
9
10  CODE_DESC:  dd   0x0000FFFF
11              dd   DESC_CODE_HIGH4
12
13  DATA_STACK_DESC: dd   0x0000FFFF
14                   dd   DESC_DATA_HIGH4
15
16 VIDEO_DESC: dd   0x80000007;limit=(0xbffff-0xb8000)/4k=0x7
17              dd    DESC_VIDEO_HIGH4  ;此时 dpl 为 0
18
19  GDT_SIZE   equ   $ - GDT_BASE
20  GDT_LIMIT  equ   GDT_SIZE - 1
21  times 60 dq 0            ; 此处预留 60 个描述符的空位
22  SELECTOR_CODE equ (0x0001<<3) + TI_GDT + RPL0
      ; 相当于(CODE_DESC - GDT_BASE)/8 + TI_GDT + RPL0
23  SELECTOR_DATA equ (0x0002<<3) + TI_GDT + RPL0    ; 同上
24  SELECTOR_VIDEO equ (0x0003<<3) + TI_GDT + RPL0   ; 同上
25
26  ;以下是 gdt 的指针，前 2 字节是 gdt 界限，后 4 字节是 gdt 起始地址
27
28  gdt_ptr  dw  GDT_LIMIT
29           dd  GDT_BASE
30   loadermsg db '2 loader in real.'
31
32   loader_start:
33
34 ;------------------------------------------------------------
35 ;INT 0x10    功能号:0x13    功能描述:打印字符串
36 ;------------------------------------------------------------
37 ;输入:
38 ;AH 子功能号=13H
39 ;BH = 页码
40 ;BL = 属性(若 AL=00H 或 01H)
41 ;CX=字符串长度
42 ;(DH、DL)=坐标(行、列)
43 ;ES:BP=字符串地址
44 ;AL=显示输出方式
45 ;  0——字符串中只含显示字符，其显示属性在 BL 中
       ;显示后，光标位置不变
46 ;  1——字符串中只含显示字符，其显示属性在 BL 中
       ;显示后，光标位置改变
47 ;  2——字符串中含显示字符和显示属性。显示后，光标位置不变
48 ;  3——字符串中含显示字符和显示属性。显示后，光标位置改变
49 ;无返回值
50   mov   sp, LOADER_BASE_ADDR
51   mov   bp, loadermsg       ; ES:BP = 字符串地址
52   mov   cx, 17              ; CX = 字符串长度
53   mov   ax, 0x1301          ; AH = 13,  AL = 01h
54   mov   bx, 0x001f          ; 页号为 0(BH = 0) 蓝底粉红字(BL = 1fh)
55   mov   dx, 0x1800
56   int   0x10                ; 10h 号中断
57
58 ; --------------------   准备进入保护模式   ------------------------------
59 ;1 打开 A20
60 ;2 加载 gdt
61 ;3 将 cr0 的 pe 位置 1
62
63
64   ;----------------  打开 A20  ----------------
65   in al,0x92
```

```
66     or al,0000_0010B
67     out 0x92,al
68
69     ;-----------------  加载 GDT  ----------------
70     lgdt [gdt_ptr]
71
72
73     ;-----------------  cr0 第 0 位置 1  ----------------
74     mov eax, cr0
75     or eax, 0x00000001
76     mov cr0, eax
77
78     jmp dword SELECTOR_CODE:p_mode_start    ; 刷新流水线
79
80
81 [bits 32]
82 p_mode_start:
83     mov ax, SELECTOR_DATA
84     mov ds, ax
85     mov es, ax
86     mov ss, ax
87     mov esp,LOADER_STACK_TOP
88     mov ax, SELECTOR_VIDEO
89     mov gs, ax
90
91     mov byte [gs:160], 'P'
92
93     jmp $
```

代码不到 100 行，里面的注释还是比较详尽的，基本上能看懂，所以就不逐行解释了，下面我就部分重点给大家介绍下。

前 2 行不用多说了，和之前版本的 loader 差不多。

第 3 行“LOADER_STACK_TOP equ LOADER_BASE_ADDR”，这个 LOADER_STACK_TOP 是用于 loader 在保护模式下的栈，它等于 LOADER_BASE_ADDR，其实这是 loader 在实模式下时的栈指针地址。只不过进入保护模式后，咱们得为保护模式下的 esp 初始化，所以用了相同的内存地址作为栈顶。LOADER_BASE_ADDR 的值是 0x900，这是 loader 被加载到内存中的位置，在此地址之下便是栈。

全局描述符表 GDT 只是一片内存区域，里面每隔 8 字节便是一个表项，即段描述符。我们这里定义段描述符的方式很简单直接，是将描述符拆成高 4 字节和低字节两部分，分别定义。这不是定义段描述符的固定方式，如果您愿意也可以直接定义 8 字节的数据，甚至也可用两两字节的方式去拼段描述符，风格不限，纯属个人喜好。咱们这里用的是 dd 指令来定义它们的，dd 是伪指令，意为 define double-word，即定义双字变量，一个字是 2 字节，所以双字就是 4 字节数据。程序编译后的地址是从上到下越来越高的，所以，下面用 dd 定义的数据地址要高于上面的 dd 所定义的数据地址，也就是说，上面的 dd 是定义的段描述符的低 4 字节，下面的 dd 是段描述符的高 4 字节。话说我都觉得有点啰嗦了，感觉说得没法再清楚了，写书真地不容易啊，总怕说得不清楚。

第 6～17 行是在构建全局描述符表，并直接在里面填充段描述符。GDT 的起始地址是标号 GDT_BASE 所在的地址。这里我们事先定义了 3 个有用的段描述符，注意啦，我说的是“有用的”，因为第 0 个段描述符没用。从第 1 个到第 3 个，分别是代码段描述符 CODE_DESC、数据段和栈段描述符 DATA_STACK_DESC、显存段描述符 VIDEO_DESC，它们三个之间都是间隔 8 字节大小，也就是每个段描述符的大小。别忘记了，我们前面说过，GDT 中的第 0 个描述符不可用，所以第 7～8 行，直接将段描述符的高 4 字节和低 4 字节，分别用 dd 定义为 0。

段描述符的低 4 字节还是比较容易定义的，其中的低 2 字节是段界限的 0～15 位，高 2 字节是段基址的 0～15 位。拿第 10～11 行的段描述符 CODE_DESC 举例，第 1 个“dd 0x0000FFFF”，这是段描述符的低 4 字节。其中低 2 字节的 FFFF 是段界限的第 0～15 位，高 2 字节的 0000 是段基址的第 0～15 位。忘记的话，赶紧参看图 4-5 段描述符的格式。

段描述符的高 4 字节相对麻烦一些，所以在 boot.inc 中直接将段描述符的高 4 位提前定义好，到 loader.S 中直接用就行啦。例如第 11 行的 dd DESC_CODE_HIGH4，这就是在定义代码段描述符的高 4 字节。

DESC_CODE_HIGH4 定义在代码 4-2 的 boot.inc 中，前面在说 boot.inc 时已经讲过这个冗长的表达式啦。

DATA_STACK_DESC 是数据段和栈段的段描述符，我们这里数据段和栈段共同使用一个段描述符，这当然是可以的，因为栈段也是数据段。其定义的原理和 CODE_DESC 一样。按理说，栈应该是向下扩展的，数据段是向上扩展的，一个段描述符只能定义一种扩展方向，type 字段中的 e 要么是 0（向上扩展），要么是 1（向下扩展）。栈也能用向上扩展的数据段吗？当然可以，只不过在这种情况下，栈段的段界限按照数据段的规则来检查了。段描述符中的各字段只是用来供 CPU 检查的，CPU 不知道此段是用来干什么的，只有用此段的人才知道。栈段向下扩展，是指栈指针 esp 指向的地址逐渐减小，不过那是 push 指令的作用，和段描述符的扩展方向无关，此扩展方向是用来配合段界限的，CPU 在检查段内偏移地址的合法性时，就需要结合扩展方向和段界限来判断。而且，用向上扩展的数据段做栈段，比用向下扩展的段更容易。我们还是挑简单的来，直接用普通的数据段做栈段，所以 type 中的 e 为 0。

下面说说显存段描述符 VIDEO_DESC。还记得实模式下的内存布局吗？赶紧看看表 1-1，其中用于文本模式显示适配器的内存地址是 0xb8000～0xbffff，内存地址 0xc0000 显示适配器 BIOS 所在区域。由于我们只支持文本模式的输出，所以为了方便显存操作，显存段不采用平坦模型。我们直接把段基址置为文本模式的起始地址 0xb8000，段大小为 0xbffff-0xb8000=0x7fff，段粒度为 4k，因而段界限 limit 等于 0x7fff /4k=7。具体见第 16～17 行，这里不再多说，大家对照着段描述符的格式验证下。

第 19～20 行，先是通过地址差来获得 GDT 的大小，进而用 GDT 大小减 1 得到了段界限，这是为加载 GDT 做准备。

第 21 行纯粹是为了将来往 GDT 中添加其他描述符，提前保留空间而已。以后我们还要往 GDT 中继续塞中断描述符表 IDT 和任务状态段 TSS 描述符。dq 用来定义了 8 字节数据，即 define quad-word，定义 4 字，即 8 字节。所以用 times 60 dq 0 提前预留 60 个描述符空位。其实也不用保留那么多，就当是为了方便扩展吧，万一哪天用到了就省事了。哦，times 是 nasm 提供的伪指令，用来重复执行 times 后面的表达式，相当于是个循环，只不过“直接”执行此循环的不是 CPU，而是编译器 nasm。指令格式是：

times 循环次数表达式

第 22～24 行是在构建代码段、数据段、显存段的选择子，按照图 4-8 选择子的格式来构造。有没有疑问，哎？怎么没有栈段的选择子。有啊，其实用的就是数据段选择子，因为数据段和栈段是同一个段描述符。一会儿在加载 ss 段寄存器选择子的时候，大家就清楚了。

第 28～29 行是定义全局描述符表 GDT 的指针，此指针是 lgdt 加载 GDT 到 gdtr 寄存器时用的，还记得 lgdt 指令的格式吗？

lgdt48 位内存数据

这 48 位内存数据的前 16 位是 GDT 以字节为单位的界限值，也就是 GDT 大小减 1。后 32 位是 GDT 的起始地址。

第 30 行就是定义个字符串，用来“显示”一下咱们要进入保护模式了。其实它还是在实模式下打印的，用的还是 BIOS 中断，此中断在 MBR 中已经介绍过啦。以后咱们就不用再 BIOS 中断了，这次就当是个告别吧。

第 52 行的 BIOS 调用中，利用 int 0x10 打印字符串的功能，cx 寄存器是字符串的长度，这是 int 0x10 的参数。“2 loader in real.”的长度是 17。

第 55 行的“mov dx， 0x1800”，其中行数 dh 为 0x18，列数 dl 为 0x00。这也是 int 0x10 的参数。由于在文本模式下的行数是 25 行，即 0～24 行，所以 0x18 的十进制为 24，即最后一行，所以，“2 loader in real.”将出现在最后一行的行首。

第 58～76 行是进入保护模式的三个步骤。分别如下。

（1）打开 A20 地址线。

（2）在 gdtr 寄存器中加载 GDT 的地址及偏移量（界限值）。

（3）将 cr0 寄存器的 pe 位置 1。

其中第（2）步，也就是 70 行的“lgdt [gdt_ptr]”，gdt_ptr 是前面已经介绍过的 GDT 地址指针变量，其值

的前 16 位是 GDT 界限值，后 32 位是 GDT 的起始地址。gdt_ptr 本身是个地址，所以要用中括号[]括起来，表示在地址处取值。该值才是 lgdt 的参数。此指令执行完后，GDT 就加载成功啦，其中的描述符如图 4-11 所示。

```
<bochs:5> info gdt
Global Descriptor Table (base=0xc0000900, limit=31):
GDT[0x00]=??? descriptor hi=0x00000000, lo=0x00000000
GDT[0x01]=Code segment, base=0x00000000, limit=0xffffffff, Execute-Only
, Non-Conforming, Accessed, 32-bit
GDT[0x02]=Data segment, base=0x00000000, limit=0xffffffff, Read/Write,
Accessed
GDT[0x03]=Data segment, base=0xc00b8000, limit=0x00007fff, Read/Write,
Accessed
You can list individual entries with 'info gdt [NUM]' or groups with 'i
nfo gdt [NUM] [NUM]'
```

▲图 4-11　GDT 中的描述符

这里面只列出了 4 个段描述符，即 0x00～0x03。这里只列出 4 个，并不是因为 CPU 知道咱们定义了 4 个描述符，CPU 并不知道 GDT 中哪些描述符是咱们定义的，内存中的随机值也许会让某些空描述符恰好凑成正确的描述符。之所以列出了咱们定义的全部描述符，是因为 lgdt 加载时，参数的前 16 位是 GDT 界限值，是它来告诉 CPU 只显示（界限值+1/段描述符大小 8 字节）个描述符，所以段描述符在定义时，一定要保证它们在 GDT 中是连续的。

第 74～76 行，也就是上面进入保护模式的第（3）步，将 PE 位置 1 后，从此便进入了保护模式的大门，此位相当于保护模式的开关。图 4-12 所示是启用 PE 位的前后对比。

```
<bochs:3> creg
CR0=0x60000010: pg CD NW ac wp ne ET ts em mp pe        PE位初值为0
CR2=page fault laddr=0x00000000
CR3=0x000000000000
    PCD=page-level cache disable=0
    PWT=page-level write-through=0
CR4=0x00000000: smap smep osxsave pcid fsgsbase smx vmx osxmmexcpt osfxsr pce pge mce pae
pse de tsd pvi vme
EFER=0x00000000: ffxsr nxe lma lme sce
<bochs:4> n                      以下三步将CR0寄存器PE位置为1
Next at t=17843182
(0) [0x000000000b39] 0000:0b39 (unk. ctxt): mov eax, cr0              ; 0f20c0
<bochs:5>
Next at t=17843183
(0) [0x000000000b3c] 0000:0b3c (unk. ctxt): or eax, 0x00000001        ; 6683c801
<bochs:6>
Next at t=17843184
(0) [0x000000000b40] 0000:0b40 (unk. ctxt): mov cr0, eax              ; 0f22c0
<bochs:7>
Next at t=17843185
(0) [0x000000000b43] 0000:00000b43 (unk. ctxt): jmp far 0008:0b48         ; ea480b0800
<bochs:8> creg
CR0=0x60000011: pg CD NW ac wp ne ET ts em mp PE        PE位现为1
CR2=page fault laddr=0x00000000
CR3=0x000000000000
    PCD=page-level cache disable=0
    PWT=page-level write-through=0
CR4=0x00000000: smap smep osxsave pcid fsgsbase smx vmx osxmmexcpt osfxsr pce pge mce pae
pse de tsd pvi vme
EFER=0x00000000: ffxsr nxe lma lme sce
<bochs:9>
```

▲图 4-12　CR0 的 PE 位

如图 4-12 所示，creg 是 bochs 中用来查看控制寄存器的命令。在 bochs 中，控制寄存器和状态寄存器中的相应位若为 1，bochs 会将该名称以大写来表示。所以，在执行中间框框中的三个步骤之前，cr0 寄存器的值为 0x60000010，pe 位是 0，名称是小写。注意啦，寄存器的值是十六进制的，其中的一位等于 4 位二进制。右边第 2 位的 1 是 CR0 寄存器第 4 位，参见 CR0 寄存器的结构，其为 ET 位，由于其值是 1，所以是大写 ET。在执行框框中的步骤之后，CR0 寄存器的值为 0x60000011，PE 位是 1，所以是大写 PE。

咱们先把话锋转一转，说点小事儿。

loader.S 中第 4 行的“jmp loader_start”，其机器码是 E91702，共 3 字节大小。其中的 E9 是操作码，1702 是操作数，由于是小端字节序，所以其十六进制是 0x217，这是 16 位相对近转移。此指令直接跳过 GDT 定义相关部分，直接到第 32 行。第 32 行的标号 loader_start 在文件内的地址是“jmp loader_start”的 3 字节机器码+4 个段描述符大小+预留的 60 个描述符大小+gdt_ptr 的 6 字节+loadermsg 的 17 个字节=3+32+480+6+17=538=0x21a。再加上 loader 被加载到的地址 0x900，在内存中的实际地址为 0x900+0x21a=0xb1a。

如果不包括第 4 行的 jmploader_start，那么 loader_start 的地址将是 0xb1a-3=0xb17。也就是说，如果把 mbr 中跳入 loader 的语句 jmp LOADER_BASE_ADDR，改成 jmp LOADER_BASE_ADDR+ 0xb17，其结果也是一样的，直接跳转到 loader.S 中的 loader_start。重点来了，也就是说，loader.S 中开头第 4 行的 jmploader_start 是可以不要的。

接下来咱们摊上大事啦。

见代码 4-3 的第 78～82 行。

```
……
78    jmp dword SELECTOR_CODE:p_mode_start     ; 刷新流水线
79
80
81 [bits 32]
82 p_mode_start:
……
```

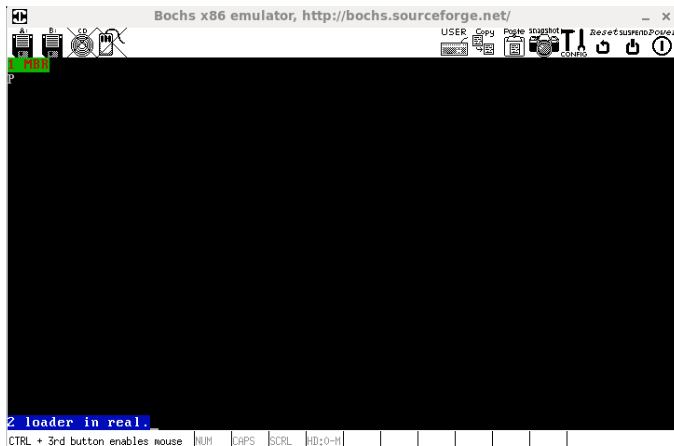
第 78 行跳入的地址是第 82 行的 p_mode_start，在第 78 行和第 82 行之间没有任何指令或数据，在没有任何“间隔阻拦”的情况下为何还要用跳转指令呢？去掉它行吗？还真不行，去掉它之后，程序就运行出错了。大伙儿不信自己试试看。也许您已经注意到了，本行代码后面的注释：刷新流水线。这是啥意思呢？

流水线是 CPU 为提高效率而采取的一种工作方式，CPU 将当前指令及其后面的几条指令同时放在流水线中重叠执行。由于在实模式下时，指令按照 16 位指令格式来译码，第 78～82 行既有 16 位指令，又有 32 位指令，所以流水线把 32 位指令按照 16 位译码就会出错。解决这问题的方法就是用无跳转指令清空流水线。

各位看官，有关流水线这里先点到为止，后面 4.4 和 4.5 两节专门讲这些体系结构相关的内容，其中主要说的就是流水线。不过咱们做事要有始有终，这里先把剩下的代码说完。

第 83～89 行，是用选择子初始化成各段寄存器。

第 90 行，是往显存第 80 个字符的位置（第 2 行首字符的位置）写入字符 P。默认的文本显示模式是 80*25，即每行是 80 个字符（0～79），每个字符占 2 字节，故传入偏移地址是 80*2=160。显存中每个字符的低字节是字符的 ASCII 值，高字节是属性位。这里咱们没有传入属性值，便会默认为黑底白字。程序执行效果如图 4-13 所示。



▲图 4-13　进入保护模式

图 4-13 左下角的字符串“2 loader in real”是在实模式下用 BIOS 中断 0x10 打印的。左上角第 2 行的字符'P'，这是咱们在保护模式下输出的。一个程序历经两种模式，各模式下都打印了字符，为了区别实模式下的打印，所以字符串中含有“inreal”。

4.4 处理器微架构简介

了解处理器内部硬件架构，有助于理解软件运行原理，因为这两者本身相辅相成，相互依存。就像枪

和狙击手，枪的操作和外形设计都要根据人体工学，让人不仅操作容易，而且携带也要轻便，做到能随时射出子弹击中目标，枪赋予了人的使命。反过来，狙击手要根据枪的操作和外形设计，找到发挥其最大命中率的方法，做到人枪合一，人赋予了枪的生命。

4.4.1　流水线

记得 2002 年年末我在中关村攒了台电脑，当时的最新的桌面 CPU 是奔 4-2.4b，我一咬牙就买它了，当时可花了兄弟我 1600 元呢，话说当时一个月只挣 1500 元。其中 2.4 是指 CPU 的主频是 2.4G。您看，对于 CPU 的选择，我当时只关注主频，根本不关注其他参数，其实主要是不懂其他参数，哈哈。后来才懂了 CPU 中另一个非常重要的技术，这就是流水线。

说到流水线，也许您脑子中马上联想到：工厂里一个穿着蓝色工作服的工人，站在像传送带一样的工作台前紧张而有节奏的工作。也许您会问，你怎么知道我是这么想的？哎……电视里都这么演的……没错，这就是流水线。可是这个例子毕竟太遥远，并不是所有同学都在工厂里做过流水线工人，咱们还是拿生活中的例子说事。

大伙儿都有租房子的经历吗？如果没有也没关系，重点不是租房子，我要说的是像我等北漂一族，为了充分利用房子里狭小的空间，肯定用锤子往墙上砸过钉子，您懂的，可以挂东西嘛。

从宏观上看，砸钉子可以分两个步骤。

（1）取钉子。

（2）砸钉子。

假设每一步都占用 1 秒，钉钉子这两个步骤下来是 2 秒，如果顺序执行这两个步骤，一分钟可以砸 30 个钉子。过程见表 4-12。

表 4-12　　顺序执行

第 1 秒	第 2 秒	第 3 秒	第 4 秒	第 n-1 秒	第 n 秒
取钉子	砸钉子	取钉子	砸钉子	取钉子	砸钉子
砸入第一个钉子		砸入第二个钉子		砸入第 n/2 个钉子	

以上是以串行顺序的方式来砸钉子，不过这种串行的效率实在有限，如果改为并行的方式砸钉子，效率必然大大提高。

生活中处处是并行的例子，最为典型的并行系统就是咱们的身体。比如心脏在为身体泵血的时候，肺同时在保持呼吸为身体供氧，小肠也同时在蠕动，为人体汲取养分。这些器官的活动都是并行的，并不是在心脏跳动完之后，小肠再蠕动，它们的工作是彼此独立无关联的。人体内部的器官虽然是在并行工作，但它们作为一个整体——人，却同一时间只能做好一件事，所以一心不能二用。

如果以“并行”的方式，也就是同时砸钉子，那得多增加人手才行，一个人同时只能做一件事，并且要么拿钉子，要么砸钉子。下面给砸钉子的工作增加个人手，专门给取钉子，这样一个人取钉子，另一个人专门来砸钉子，取钉子和砸钉子的工作重叠进行，过程见表 4-13。

表 4-13　　重叠执行

钉子	第 1 秒	第 2 秒	第 3 秒	第 4 秒	第 n-1 秒	第 n 秒
第一个钉子	取钉子	砸钉子				
第二个钉子		取钉子	砸钉子			
第三个钉子			取钉子	砸钉子		

上面的并行我加了引号，因为它并不是真正地并行，这是在重叠执行。重叠的意思是说在同一时间内同时完成两个钉子的部分工序，拿第 2 秒来说，第一行的“砸钉子”是砸的第一个钉子，第二行的“取钉子”取的是第二个钉子，做的并不都是第一个钉子的工序。真正的并行是两个人自己取自己的钉子，然后自己砸自己的钉子，各干各的。而我们的例子中，取钉子的人只会取钉子，砸钉子的人也只会砸钉子。

增加了一个人手之后，除第 1 秒外，每一秒都有“砸钉子”的动作，所以第一分钟内可以砸入 59 根

钉子，在第二分钟以后，每分钟能砸 60 个。

表 4-13 的过程便是一个流水线的执行过程，由于砸钉子分为两个步骤，所以以上流水线称为二级流水线。

这种情况在 CPU 中也是一样的，指令执行单元 EU 是执行指令的唯一部件，一次只能执行一个指令，单核 CPU 的情况下，只有一个指令处于执行中。CPU 中的各部分也是同时只能做一件事，但它们就像身体器官一样，也是在并行工作，相当于多个“人手”。CPU 的指令执行过程分为取指令、译码、执行三个步骤。每个步骤都是独立执行的，CPU 可以一边执行指令，一边取指令，一边译码。CPU 中的时序不是秒，对 CPU 来说，秒就是天文数字。它的时序是时钟周期。按照这三个步骤，其三级流水线见表 4-14。

表 4-14 三级流水线

指令	周期 1	周期 2	周期 3	周期 4	周期 5	周期 6
第一条指令	取指	译码	执行			
第二条指令		取指	译码	执行		
第三条指令			取指	译码	执行	

以上在第 2 周期后，都有指令在执行，这是最基本的流水线啦。为了更好地理解以后的分支预测，在此提醒一下大伙儿：虽然在一个时钟周期内 CPU 同时干了三件事，但一定要清楚，这三件事不属于同一个指令，是三个指令重叠在一起了，这和砸钉子的流水线是一样的道理。同时完成的是当前指令的第三步、下一条指令的第二步、第三条指令的第一步。CPU 中每条指令必须经过取指、译码、执行三步才算完成。就拿表 4-14 中的周期 3 来说，在这一时钟周期里，CPU 同时完成了“执行”、“译码”、“取指”三件事。其中的“执行”是执行的第一条指令，“译码”是在为第二条指令译码，取指是从内存中取出第三条指令。

在这里不得不插一句，CPU 是按照程序中指令顺序来填充流水线的，也就是说按照程序计数器 PC（x86 中是 cs：ip）中的值来装载流水线的，当前指令和下一条指令在空间上是挨着的。如果当前执行的指令是 jmp，下一条指令已经被送上流水线译码了，第三条指令已经被送上流水线取指啦。仔细想想看，其实这个流水线没用了，因为 CPU 早已经跳到别处去执行了，第二、三条指令用不上了，所以 CPU 在遇到无条件转移指令 jmp 时，会清空流水线。

回到正题，其实，流水线还是有优化空间的，表 4-4 才 3 级流水线，而奔腾 CPU 可是 32 级流水线呢。CPU 指令三个步骤中，只有“执行”这一步才是最重要的，想办法让此步骤的周期更短才是王道。也就是说，执行周期越短，CPU 所执行指令的数量越多，效率也就越高，但流水线级数肯定越多。解决问题的办法是，将每一步操作再继续划分成粒度更细的微操作。为了说明问题，咱们还是拿砸钉子举例。仔细分析下砸钉子的两个步骤：第 1 个步骤取钉子，可以分解为取钉子、定位。第 2 个步骤钉钉子，可以分解为瞄准，砸钉子。全部过程是：取出钉子后，确定钉入的位置，拿起锤子，瞄准，砸下去。最终起作用的步骤是砸钉子，前三小步都是在做准备，砸钉子这步时间越短越好。由于每个大步骤耗时 1 秒，现假设每个小步骤耗时 0.5 秒。按小步骤重新安排流水线，情况见表 4-15。

表 4-15 四级流水线

钉子	第 0.5 秒	第 1 秒	第 1.5 秒	第 2 秒	第 2.5 秒	第 3 秒	第 3.5 秒
第一个钉子	取钉子	定位	瞄准	砸钉子			
第二个钉子		取钉子	定位	瞄准	砸钉子		
第三个钉子			取钉子	定位	瞄准	砸钉子	
第四个钉子				取钉子	定位	瞄准	砸钉子

从第 2 秒后，每 0.5 秒就会有一个砸钉子的动作，所以在以后的每分钟内，都会钉入 120 个钉子，速度又提高了很多。这就是将指令拆分成多个微操作后的效率提升。

流水线是 CPU 提高效率的一种出路，以后介绍的各种优化方法，其实都是围绕如何让流水线更加有效而展开的。走，咱们继续看看 CPU 工程师们还做了哪些努力。

4.4.2　乱序执行

乱序执行，是指在 CPU 中运行的指令并不按照代码中的顺序执行，而是按照一定的策略打乱顺序执行，也许后面的指令先执行，当然，得保证指令之间不具备相关性。

举个简单的例子，比如如下两行代码就无法乱序执行。

```
1 mov eax, [0x1234]
2 add eax, ebx
```

第 2 行的 add 加法，需要知道 eax 的值，但 eax 的值需要在第 1 行中的 mov 操作后才能确定，而且内存访问相对来说非常慢，第 2 步不得不等待第 1 步完成后才能进行。所以只能是先执行第 1 步，再执行第 2 步。

如果将上面第 2 步的代码修改一下，如下：

```
1 mov eax, [0x1234]
2 add ecx,ebx。
```

这样就可以在执行第 1 步内存访问后的等待中执行第 2 步啦。由于第 2 步不依赖第 1 步，所以有利于放在流水线上。

x86 最初用的指令集是 CISC（Complex Instruction Set Computer），意为复杂指令集计算机，为什么复杂呢？当初的 CPU 工程师们为了让 CPU 更加强大，不断地往 CPU 中添加各种指令，甚至在 CPU 硬件一级直接支持软件中的某些操作，以至于指令集越来越庞大笨重复杂。例如 push 指令，它相当于多个子操作的合成，拿保护模式中的栈来说，push eax 相当于：

- push 指令先将栈指针 esp 减去操作数的字长，如 sub esp,4。
- 再将操作数 mov 到新的 esp 指向的地址，如 mov [esp],eax。

这两个子操作合成了一个指令，其中每一个子操作称为微操作。

与 CISC 指令集相对应的是 RISC（Reduced Instruction Set Computer），意为精简指令集计算机。根据二八定律，最常用的指令只有 20%，但它们占了整个程序指令数的 80%。而不常用的指令占 80%，但它们只占整个程序指令数的 20%。这就是 RISC 指令集的由来，它精简保留了那些常用的指令，这些指令大多数都是不可再细分的，也就是，它们基本上都是属于微操作级别的指令啦。

所以，x86 发展到后来，虽然还是 CISC 指令集，但其内部已经采用 RISC 内核，译码对于 x86 体系来说，除了按照指令格式分析机器码外，还要将 CISC 指令分解成多个 RISC 指令。当一个“大”操作被分解成多个“微”操作时，它们之间通常独立无关联，所以非常适合乱序执行。

还是拿栈举例，如下三行代码。

```
1 mov eax , [0x1234]
2 push eax
3 call function
```

第 1 步需要内存访问，由于内存较慢，所以寻址等待过程中可以做其他事。

第 2 步的 push 指令拆分成 sub esp ，4 和 mov [esp]， eax。

第 3 步的 call 函数调用，需要在栈中压入返回地址，所以说 call 指令需要用栈指针。

由于第 2 步中的微操作 sub esp，4，可以让 CPU 知道 esp 的最新值，不用等到 mov [esp], eax 完成，第 3 步 call 指令向栈中压入返回地址的操作就可以执行了。故第 2 步未执行完就开始第 3 步的执行了，也许第 3 步先于第 2 步完成。

总结一下，乱序执行的好处就是后面的操作可以放到前面来做，利于装载到流水线上提高效率。

4.4.3　缓存

缓存是 20 世纪最大的发明，其原理是用一些存取速度较快的存储设备作为数据缓冲区，避免频繁访问速度较慢的低速存储设备，归根结底的原因是低速存储设备是整个系统的瓶颈，缓存用来缓解“瓶颈设备”的压力。

第 3 章介绍实模式下的寄存器时，举了一个浏览器访问网页的例子，里面有 12 步，几乎步步都用到了缓存。

不过，这些缓存都是在内存 DRAM 中实现的，即动态随机访问存储器，究其原因是数据要么在数据库中，要么在硬盘上，其速度肯定比内存慢。选作缓存的存储设备，其存取速度肯定是比其原有存储设备更快，否则失去了缓存的意义。相对于 CPU 来说，DRAM 太慢了，如果也要用它来做 CPU 的缓存，反而是拖了后腿，不如不用。

CPU 为什么要用缓存？因为待执行的指令和相关数据存储在低速的内存中，让 CPU 这种高速设备等待慢速的内存，着实太浪费 CPU 资源了。人们根本无法容忍 CPU 如此“漫长”的“浪费”，所以需要用一个比内存更快的存取设备做缓冲区，尽量和 CPU 一个速度，让 CPU 不要等待。于是 SRAM 成了 CPU 的救世主，成为 CPU 和内存之间数据缓存的不二之选。

前面在介绍实模式下的寄存器时，也说到了 CPU 中的缓存。CPU 中有一级缓存 L1、二级缓存 L2，甚至三级缓存 L3 等。它们都是 SRAM，即静态随机访问存储器，它是最快的存储器啦。之所以把 SRAM 和寄存器放到一块说，是因为很多同学在感观上觉得寄存器是 CPU 直接使用的存储单元，所以寄存器比 SRAM 更快。其实它们在速度上是同一级别的东西，因为寄存器和 SRAM 都是用相同的存储电路实现的，用的都是触发器，它可是工作速度极快的，属于纳秒级别。至于触发器是什么，这已属于硬件范畴，这里就不深究了，有兴趣的读者请自行调研吧。

有哪些东西可以被缓存呢？无论是程序中的数据，还是指令，在 CPU 眼里全是一样形式的二进制 01 串，没有任何区别，都是 CPU 待处理的“数据”。所以我们眼中的指令和数据都可以被缓存到 SRAM 中。

什么时候能缓存呢？可以根据程序的局部性原理采取缓存策略。局部性原理是：程序 90%的时间都运行在程序中 10%的代码上。

局部性分为以下两个方面。

一方面是时间局部性：最近访问过的指令和数据，在将来一段时间内依然经常被访问。

另一方面是空间局部性：靠近当前访问内存空间的内存地址，在将来一段时间也会被访问。

举一个典型的例子，我们在用高级语言写程序时，经常会写到这样的循环嵌套代码，如：

```
int array[100][100];
int sum = 0;
…
数组 array 元素被赋值，略
…
for (int i=0, i<100,i++) {
     for(int j=0;j<100,j++) {
     sum+=array[i][j];
   }
}
```

以上是将二维数组中的所有元素相加求和的代码。循环中经常被用到的地址是 sum 所在的地址，经常被用到的指令是加法求和指令，这是在时间上的局部性。未来要访问的地址是与当前访问地址&array[i][j]相邻的地址&array[i][j+1]，它们之间只差一个整型变量的大小，这是空间上的局部性的。（当然，这些局部性都是编译器编译的结果，编译器就是这样安排的。）CPU 利用此特性，将当前用到的指令和当前位置附近的数据都加载到缓存中，这就大大提高了 CPU 效率，下次直接从缓存中拿数据，不用再去内存中取啦。

当然，上面说的是理想的状态，如果缓存中没有相应的数据，还是要去内存中加载，然后再放到缓存中。

4.4.4　分支预测

人在道路的分岔口时要预测哪条路能够到达目的地，面对众多选择时，计算机也一样要抉择，毕竟计算机的运行方式是以人的思路来设计的，计算机中的抉择其实就是人在抉择。

CPU 中的指令是在流水线上执行。分支预测，是指当处理器遇到一个分支指令时，是该把分支左边的指令放到流水线上，还是把分支右边的指令放在流水线上呢？

如 C 语言程序中的 if、switch、for 等语言结构，编译器将它们编译成汇编代码后，在汇编一级来说，这些结构都是用跳转指令来实现的，所以，汇编语言中的无条件跳转指令很丰富，以至于称之为跳转指令“族”，多得足矣应对各种转移方式。

举个例子，如下面的测试代码。

```
1 void main () {
2     int i = 0;
3     while (i < 10) {
4        i++;
5     }
6 }
```

里面的 while 结构，就是执行了 10 次 i++。我们来看一下 while 结构是如何翻译成汇编语言的。

gcc -S -o ～/test/while.S ～/test/while.c 回车，这样 gcc 就将 while.c 编译成了汇编代码 while.S。其中的参数-S 是编译到汇编语言，不进行汇编和链接。

查看下～/test /while.S 文件，cat -n ～/test/while.S 回车。

```
 1     .file"while.c"
 2     .text
 3 .globl main
 4     .type    main, @function
 5 main:
 6     pushl    %ebp
 7     movl     %esp, %ebp
 8     subl     $16, %esp
 9     movl     $0, -4(%ebp)
10     jmp .L2
11 .L3:                                 ;此处是 while 的循环体
12     addl     $1, -4(%ebp)
13 .L2:                                 ;此处是 while 循环条件表达式
14     cmpl     $9, -4(%ebp)
15     jle .L3
16     leave
17     ret
18     .size    main, .-main
19     .ident   "GCC: (GNU) 4.4.6 20120305 (Red Hat 4.4.6-4)"
20     .section .note.GNU-stack,"",@progbits
```

这个生成的汇编语言并不是我们熟悉的 Intel 语法，而是 AT&T 语法，如果此时您觉得太陌生也不要慌张，因为在后面的章节我们会专门说到此类语法，现在先抛出来和大家预预热。

本来打算只列出第 9～15 行的，但考虑到本身才 20 行，干脆就全贴出来了，简要说明一下，前 4 行用于声明代码段，导出 main 函数符号。第 5 行是 main 函数的起始地址，高级语言中的函数名在汇编语言中只是个符号，而符号便是地址，这就是很多教科书上都说函数名是地址的原因。话说数组也同理，数组名在汇编语言中也是个标号地址，所以数组名也是地址。局部变量是在栈中分配空间的，所以第 6～8 行是在创建堆栈框架，也就是为局部变量 i 在栈中分配空间，-4（%ebp）便是指局部变量 i。堆栈框架以后会说到。咱们主要是看第 9～15 行。

第 9 行是为变量 i 赋值为 0。AT&T 语法中，寄存器前要用%来指示，立即数前要用$来指示。-4（%ebp）表示内存地址“ebp 寄存器的值减 4”处的内存内容，相当于 Intel 汇编语法形式[ebp – 4]。AT&T 语法中是源操作数在左，目的操作数在右，和 Intel 语法相反。所以第 9 行是将 0 送入了变量 i 所在的栈空间。

第 10 行就是简单的无条件跳转，直接进入 while 循环结构的条件表达式判断，也就是第 13 行。

第 14 行就是 while 括号中的条件表达式，用变量 i 的值和立即数 9 做比较。

第 15 行的 jle 意思是若第 14 行的比较结果是小于等于 9，则跳到 11 行，继续执行第 12 行的加法。可见第 11～12 行是循环体。

程序执行流是由第 15 行跳到第 11 行，这样组成了循环结构的回路。

程序执行 while 循环后就结束了，所以局部变量 i 所在的栈空间要被回收，第 16 行的指令 leave 用于堆栈框架的回收工作。

第 17 行是 main 函数退出。由于 main 也是被调用的，所以 gcc 显式地帮咱们加了个 ret 以示退出，为什么 main 也是由别人调用的？这个在加载用户程序时咱们会说到的。

上面的第 15 行 jle 指令就是程序中的分支结构。我们花了“大力气”讲述了程序流的分支，这并不是浪费力气。类似这样的分支结构很多，它们只有两种结果，要么转移到这一边，要么转移到那一边。分支结构虽然让程序更加灵活多样，但这却成了 CPU 执行效率的诟病。这是怎么回事呢？

之前说流水线的时候，我和大家强调了两次“重叠”，即同一时间周期内完成的是当前指令的执行，下一条指令的译码，第三条指令的取指。其中最重要的是“执行”，指令只有执行了，才真正是泼出去的水，收不回来了。另外的译码和取指并不重要，首先它们并不是执行，其次它们也不属于当前指令，当前指令的“取指”和“译码”早就在前两个周期内完成了。

不知道您注意到了没有，拿表 4-14 的周期 3 来说，这一时钟周期内的“执行”指的是当前指令的执行阶段，“取指”和“译码”这两个工序分别隶属于未来要执行的下一条指令和下下一条指令。想到这里不禁要有个疑问，这两个未来的指令，CPU 是如何确定的？如果程序一直是顺序执行的，未来无论多少条指令都可以轻易得到，都可以提前放到流水线上。可是，程序是有分支啊，到底该把哪个分支的指令放到流水线上呢？

流水线是有效提升 CPU 效率的方式，但流水线最大的问题是程序中的分支结构，如何把握好转移的方向，才是使流水线保持高效的关键，因为如果流水线上的指令放错了的话，必须要清空那些已经在流水线上的指令，一定不能执行错误的指令。随着流水线级数越多，要清空的指令也将越多，清空流水线的代价就越大，这严重影响 CPU 效率。

当遇到一个分岔口时，是往左走，还是往右走呢？对于这种分支情况，就需要预测出哪一侧的指令将被执行，然后将预测出的那一分支上的指令放入流水线。从统计学的角度来看，某些事情一旦出现，下一次出现的机率还会很大。纵观历史，很多事情都是在重复地发生，很多伟人都拿这些历史样本来预测未来发生的事情。这个说的有点悬乎了，说点简单的，比如现在是葡萄收获的季节，今天刚吃了葡萄，很好吃，明天后天甚至未来的几周都会继续吃葡萄，哈哈，我太爱吃葡萄了。

让我们说说预测的算法吧。

对于无条件跳转，没啥可犹豫的，直接跳过去就是了。所谓的预测是针对有条件跳转来说的，因为不知道条件成不成立。最简单的统计是根据上一次跳转的结果来预测本次，如果上一次跳转啦，这一次也预测为跳转，否则不跳。

最简单的方法是 2 位预测法。用 2 位 bit 的计数器来记录跳转状态，每跳转一次就加 1，直到加到最大值 3 就不再加啦，如果未跳转就减 1，直到减到最小值 0 就不再减了。当遇到跳转指令时，如果计数器的值大于 1 则跳转，如果小于等于 1 则不跳。这只是最简单的分支预测算法，CPU 中的预测法远比这个复杂，不过它们都是从 2 位预测法发展起来的。

分支指令所在地址	预测的分支地址	跳转统计

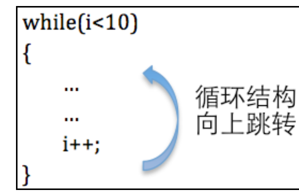
▲图 4-14　BTB 结构

算法有了，咱们看看 CPU 是如何实现预测的。

Intel 的分支预测部件中用了分支目标缓冲器（Branch Target Buffer，BTB）。其结构如图 4-14 所示。

BTB 中记录着分支指令地址，CPU 遇到分支指令时，先用分支指令的地址在 BTB 中查找，若找到相同地址的指令，根据跳转统计信息判断是否把相应的预测分支地址上的指令送上流水线。在真正执行时，根据实际分支流向，更新 BTB 中跳转统计信息。

如果 BTB 中没有相同记录该怎么办呢？这时候可以使用 Static Predictor，静态预测器。为什么称为静态呢？这是因为存储在里面的预测策略是固定写死的，它是由人们经过大量统计之后，根据某些特征总结出来的。比如，转移目标的地址若小于当前转移指令的地址，则认为转移会发生，因为通常循环结构中都用这种转移策略，为的是组成循环回路。所以静态预测器的策略是：若向上跳转则转移会发生，若向下跳转则转移不发生，如图 4-15 所示。



▲图 4-15　循环结构向上跳转

程序在实际执行转移分支指令后，再将转移记录录入到 BTB。

还记得之前反复强调的重叠吗？其实是用在这的。如果分支预测错了，也就是说，当前指令执行结果与预测的结果不同，这也没关系，只要将流水线清空就好了。因为处于执行阶段的是当前指令，即分支跳转指令。处于“译码”“取指”的是尚未执行的指令，即错误分支上的指令。只要错误分支上的指令还没到执行阶段就可以挽回，所以，直接清空流水线就是把流水线上错误分支上的指令清掉，再把正确分支上的指令加入到流水线，只是清空流水线代价比较大。

好啦各位，关于微架构这块咱们说到这就够用了，咱们当初是想解决代码 4-3 中第 78 行的清空流水线的，不能偏离目标太远啦。

4.5 使用远跳转指令清空流水线，更新段描述符缓冲寄存器

大伙应该还记得代码 4-3 中的第 78 行的无条件跳转指令吧：

```
jmp dword SELECTOR_CODE:p_mode_start
```

当时在遇到它的时候只是简单地说了一下，本节该把它严肃地说清楚了。为什么要用 jmp 远转移，是因为我们有两个问题要解决。

首先，段描述符缓冲寄存器未更新，它还是实模式下的值，进入保护模式后需要填入正确的信息。

在讲述保护模式下的寄存器扩展时，我们说到了段描述符缓冲寄存器，它首次在 80286 中出现，是为了加速段描述符中信息的访问而设的。如今的 32 位 CPU 的保护模式也依然要用到段描述符缓冲寄存器。32 位 CPU 虽然兼容实模式，但在实模式下运行时并不是变成纯 16 位的 CPU。兼容是指能够正确处理 16 位模式的程序，并不要求变成 16 位 CPU。在 16 位 CPU 中，访问内存时段基址要左移 4 位，然后再与段内偏移地址相加求和后去访问内存。而此过程在 32 位 CPU 的实模式下有所不同，即段基址左移 4 位后被送入了段描述符缓冲寄存器，随后再用此缓冲寄存器的值与段内偏移地址相加。也就是说，32 位的 CPU 已经有了缓冲寄存器，就应该充分发挥其作用。既然 CPU 实模式下的段基址每次都要左移 4 位，何不将左移 4 位后的结果放到段描述符缓冲寄存器中缓存起来，下次再有段内访问时就不用再重复计算段基址了。由于实模式下的段基址只有 20 位，所以段描述符缓冲寄存器中的低 20 位有效，用于存储段基址，其他位都为 0。

总结一下。

段描述符缓冲寄存器在 CPU 的实模式和保护模式中都同时使用，在不重新引用一个段时，段描述符缓冲寄存器中的内容是不会更新的，无论是在实模式，还是保护模式下，CPU 都以段描述符缓冲寄存器中的内容为主。实模式进入保护模式时，由于段描述符缓冲寄存器中的内容仅仅是实模式下的 20 位的段基址，很多属性位都是错误的值，这对保护模式来说必然会造成错误，所以需要马上更新段描述符缓冲寄存器，也就是要想办法往相应段寄存器中加载选择子。

其次，流水线中指令译码错误。

在默认情况下，如果未使用 bits 伪指令来设置运行环境，编译器就将代码按照 16 位实模式编译。代码 4-3，即 loader.S 中唯一的 bits 指令是在 81 行，所以 80 行之前的代码运行在实模式之下，它们是 16 位指令格式。第 81 行的[bits 32]是让编译器将此行后面的指令编译成为 32 位。因为此处已经是在保护模式下了，我们知道保护模式下的指令是 32 位，所以要编译成符合保护模式的指令格式。

我们已经知道，CPU 为了提高效率而采用了流水线，这样，指令间是重叠执行的。第 76 行之前的指令都是 16 位指令，自 76 行之后，CPU 便进入了保护模式，故第 78 行的指令已经是在保护模式下了，但它依然还是 16 位的指令，相当于处于 16 位保护模式下。为了让其使用 32 位偏移地址，所以添加了伪指令 dword，故其机器码前会加 0x66 反转前缀。而第 81 行后的代码是在[bits 32]之后，所以全是 32 位指令。

流水线的工作是这样的：在第 76 行代码执行的同时，第 78 行和之后的部分指令已经被送上流水线了，但是，段描述符缓冲寄存器在实模式下时已经在使用了，其低 20 位是段基址，但其他位默认为 0，也就是描述符中的 D 位为 0，这表示当前的操作数大小是 16 位。流水线上的指令全是按照 16 位操作数来译码的，这就坏了，人家 83 行开始的指令明明是 32 位指令，16 位和 32 位的指令都有各自不同的意义，这怎么能不出错呢？所以，如果将第 78 行的跳转指令去掉，程序将在第 83 行开始出错，原因就是 83 行的代码是 32 位指令格式，而 CPU 是将其按照 16 位指令格式来译码的，译码之后在其执行时，必然是错误的。

综上所述，解决问题的关键就是既要改变代码段描述符缓冲寄存器的值，又要清空流水线。

代码段寄存器 cs，只有用远过程调用指令 call、远转移指令 jmp、远返回指令 retf 等指令间接改变，没有直接改变 cs 的方法，如直接 mov cs，xx 是不行的。另外，之前介绍过了流水线原理，CPU 遇到 jmp 指令时，之前已经送上流水线上的指令只有清空，所以 jmp 指令有清空流水线的神奇功效。

故，用无条件远跳转指令 jmp 来解决上述两个问题将是一举两得的做法。

补充一下，代码 4-3 的第 78 行 jmp dword SELECTOR_CODE：p_mode_start，由于已经身处保护模式，所以 CPU 将此指令中的 SELECTOR_CODE 认为是选择子。因为当前段描述符缓冲寄存器中的 D 位是 0，所以操作数是 16 位，当前属于 16 位保护模式。故，在这里也可以把 dword 去掉，毕竟当前操作数大小就是 16 位。而且 p_mode_start 的地址并没有超过 16 位，用 dword 表示的 32 位地址并没有发挥其功效。这两者的区别见表 4-16。

表 4-16　　机器码对比

汇 编 代 码	实 际 指 令	机　器　码
jmp dword SELECTOR_CODE：p_mode_start	jmp far 0008：00000b4b	66ea4b0b00000800
jmp SELECTOR_CODE：p_mode_start	jmp far 0008：0b48	ea480b0800

第一行的机器码为 66ea4b0b00000800。加了伪指令 dword 后，编译器引用 32 位地址，所以加了 0x66 是指反转操作数大小前缀。

第二行的机器码为 ea480b0800，这是引用的 16 位地址。

至于操作数中的偏移地址，一个是 0xb4b，一个是 0xb48，它们之间差了 3，是由不同指令本身所占空间不同导致的，大家可以数下这两个指令的机器码大小，确实是差了 3 字节。

4.6 保护模式之内存段的保护

保护模式中的保护二字体现在哪里？其实主要体现在段描述符的属性字段中。每个字段都不是多余的。这些属性只是用来描述一块内存的性质，是用来给 CPU 做参考的，当有实际动作在这片内存上发生时，CPU 用这些属性来检查动作的合法性，从而起到了保护的作用。

本节围绕内存段来做个基本的阐述，其他方面的保护，如特权级，以后咱们再开专门的章节来细说。

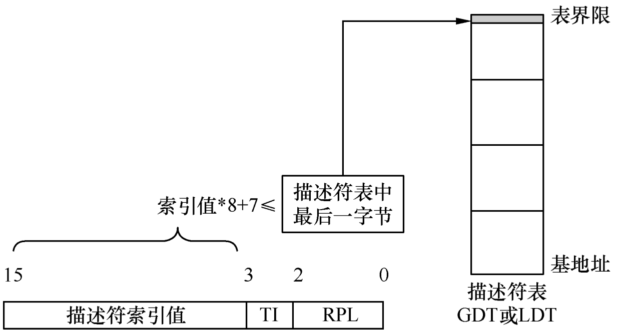
4.6.1　向段寄存器加载选择子时的保护

当引用一个内存段时，实际上就是往段寄存器中加载个选择子，为了避免出现非法引用内存段的情况，在这时候，处理器会在以下几方面做出检查。

首先根据选择子的值验证段描述符是否超越界限。

选择子的高 13 位是段描述符的索引值，第 0～1 位是 RPL，第 2 位是 TI 位。忘记其结构的同学可以参考图 4-8。首先处理器得保证选择子是正确的，判断的标准是选择子的索引值一定要小于等于描述符表（GDT 或 LDT）中描述符的个数。这就像数组下标一样，绝对不能越界。也就是说，段描述符的最后 1 字节一定要在描述符表（GDT 或 LDT）的界限地址之内。每个段描述符的大小是 8 字节，所以在往段寄存器中加载选择子时，处理器要求选择子中的索引值要满足下面表达式：描述符表基地址+选择子中的索引值*8+7 <=描述符表基地址+描述符表界限值。

检查过程如下：处理器先检查 TI 的值，如果 TI 是 0，则从全局描述符表寄存器 gdtr 中拿到 GDT 基地址和 GDT 界限值。如果 TI 是 1，则从局部描述符表寄存器 ldtr 中拿到 LDT 基地址和 LDT 界限值。有了描述符表基地址和描述符表界限值后，把选择子的高 13 位代入上面的表达式，若不成立，处理器则抛出异常。过程如图 4-16 所示。



▲图 4-16　加载选择子时的保护

在此提醒一下，GDT 中的第 0 个描述符是空描

述符，如果选择子的索引值为 0 则会引用到它。所以，不允许往 CS 和 SS 段寄存器中加载索引值为 0 的选择子。虽然可以往 DS、ES、FS、GS 寄存器中加载值为 0 的选择子，但真正在使用时 CPU 将会抛出异常，毕竟第 0 个段描述符是哑的，不可用。

段描述符中还有个 type 字段，这用来表示段的类型，也就是不同的段有不同的作用。在选择子检查过后，就要检查段的类型了。

这里主要是检查段寄存器的用途和段类型是否匹配。大的原则如下。

- 只有具备可执行属性的段（代码段）才能加载到 CS 段寄存器中。
- 只具备执行属性的段（代码段）不允许加载到除 CS 外的段寄存器中。
- 只有具备可写属性的段（数据段）才能加载到 SS 栈段寄存器中。
- 至少具备可读属性的段才能加载到 DS、ES、FS、GS 段寄存器中。

如果 CPU 发现有任意上述规则不符，检查就不会通过。以上所述可见表 4-17。

表 4-17　　段类型检查

段寄存器	代码段（X=1）		数据段（X=0）	
	只执行（R=0）	执行+可读（R=1）	只读（R=1，W=0）	读写（W=1）
CS	通过	通过	不通过	不通过
DS	不通过	通过	通过	通过
ES	不通过	通过	通过	通过
FS	不通过	通过	通过	通过
GS	不通过	通过	通过	通过
SS	不通过	不通过	不通过	通过

检查完 type 后，还会再检查段是否存在。CPU 通过段描述符中的 P 位来确认内存段是否存在，如果 P 位为 1，则表示存在，这时候就可以将选择子载入段寄存器了，同时段描述符缓冲寄存器也会更新为选择子对应的段描述符的内容，随后处理器将段描述符中的 A 位置为 1，表示已经访问过了。如果 P 位为 0，则表示该内存段不存在，不存在的原因可能是由于内存不足，操作系统将该段移出内存转储到硬盘上了。这时候处理器会抛出异常，自动转去执行相应的异常处理程序，异常处理程序将段从硬盘加载到内存后并将 P 位置为 1，随后返回。CPU 继续执行刚才的操作，判断 P 位。

注意啦，以上所涉及到的 P 位，其值由软件（通常是操作系统）来设置，由 CPU 来检查。A 位由 CPU 来设置。

4.6.2　代码段和数据段的保护

对于代码段和数据段来说，CPU 每访问一个地址，都要确认该地址不能超过其所在内存段的范围。

前面说过啦，实际段界限的值为：

（描述符中段界限+1）*（段界限的粒度大小：4k 或者 1）-1。

对于 G 位为 1 的 4k 粒度大小的段来说，其实用下面这个公式更为直接。

实际段界限大小 ＝ 描述符中段界限*0x1000+0xFFF

其中，0xFFF 是 4k（0x1000）中以 0 为起始的最后一字节。所以此公式的意义是以 0 为起始的段偏移量，即段界限。推导过程也很简单，就是将原公式展开：

（描述符中段界限+1）*4k-1=描述符中段界限*4k+4k-1 =描述符中段界限*0x1000+0xFFF。

实际的段界限大小，是段内最后一个可访问的有效地址。由于有了段界限的限制，我们给 CPU 提交的每一个内存地址，无论是指令的地址，还是数据的地址，CPU 都要帮我们检查地址的有效性。首先地址指向的数据是有宽度的，CPU 要保证该数据一定要落在段内，不能“骑”在段边界上。下面我们分情况讨论。

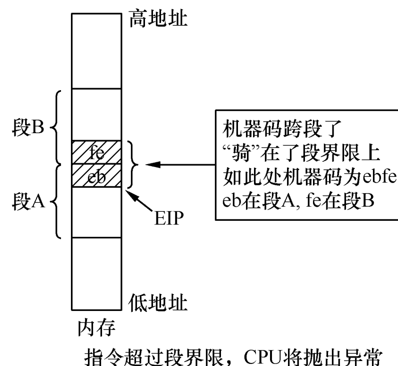
对于代码段来说，段中的“数据”是各种机器指令。有部分同学总以为只有数据段才用内存分段策略

访问，这是“误会”，这里再重申一下，在 IA32 体系结构中，访问内存就要用分段策略，这是它的宿命，逃不掉的^_^。代码段既然也是内存中的区域，所以对于代码段的访问也要用“段基址：段内偏移地址”的形式，在 32 位保护模式下，段基址存放在 CS 寄存器中，段内偏移地址，即有效地址，存放在 EIP 寄存器中。

CS：EIP 只是指令的起始地址，指令本身也是有长度的，之前我们见过各种各样的机器码，它们的长度有 2 字节的、3 字节的等，如 jmp .-2，其机器码为 ebfe，大小就是 2 字节。CPU 得确保指令“完全、完整”地任意一部分都在当前的代码段内，也就是要满足以下条件：

EIP 中的偏移地址+指令长度-1≤实际段界限大小

如果不满足条件，如图 4-17 所示，指令未完整地落在本段内，CPU 则会抛出异常。



▲图 4-17 代码段非法访问

这种边界检查对于数据段也是一样，数据也是有长度的（不同类型数据的长度不一致，这就是数据类型的作用），CPU 也要保证操作数要“完全、完整”地任意部分都要在当前数据段内。所以，数据地址也要遵循此原则：

偏移地址+数据长度-1≤实际段界限大小

同图 4-17 类似，数据段的段界限也是段内可访问的最后一个地址，所以不允许出现数据“骑”在段边界的情况，这里不再单独画图。

举个例子，假设数据段描述符的段界限是 0x12345，段基址为 0x00000000。

如果 G 位为 0，那么实际段界限便是 0x12345。如果 G 位为 1，那么实际段界限便是 0x12345*0x1000+0xFFF=0x12345FFF。如果访问的数据地址是 0x12345FFF，还要看访问的数据宽度。

若数据大小是 1 字节，如 mov ax，byte [0x12345fff]，这种内存操作一点问题都没有，数据完全在实际段界限之内。

若该数据大小是 2 字节，如 mov ax，word [0x12345fff]，这种内存操作超过了实际的段界限，数据所在地址分别是 0x12345FFF 和 0x12346000 这两个字节，CPU 会抛异常。

4.6.3 栈段的保护

前面我们在 loader.S 中用了向上扩展的数据段作为栈段，本节介绍下用向下扩展的数据段作为栈段的情况。

虽然段描述符 type 中的 e 位用来表示段的扩展方向，但它和别的描述符属性一样，仅仅是用来描述段的性质，即使 e 等于 1 向下扩展，依然可以引用不断向上递增的内存地址，即使 e 等于 0 向上扩展，也依然可以引用不断向下递减的内存地址。栈顶指针[e]sp 的值逐渐降低，这是 push 指令的作用，与描述符是否向下扩展无关，也就是说，是数据段就可以用作栈。

可能您会问了，本来以为向下扩展的段是专门给栈用的，现在又说数据段就可以用作栈，那它与向上扩展的段有什么区别？

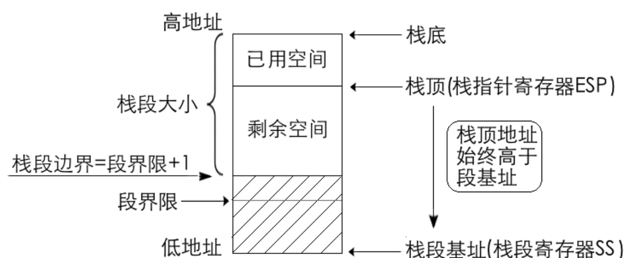
CPU 对数据段的检查，其中一项就是看地址是否超越段界限。如果将向上扩展的数据段用作栈，那 CPU 将按照上一节提到的数据段的方式检查该段。如果用向下扩展的段做栈的话，情况有点复杂，这体现在段界限的意义上。

- 对于向上扩展的段，实际的段界限是段内可以访问的最后一字节。
- 对于向下扩展的段，实际的段界限是段内不可以访问的第一个字节。

我知道这么说您可能不会明白，下面咱们具体一点。

由于栈段是向下扩展的，也许有同学觉得段界限似乎用负数更为“贴切”。但段界限本质上就是段的大小范围，范围可没有负数之说，所以段界限肯定是个正数。如果您觉得栈的段界限应该是负数的话，可能的原因是您把位于高地址处的栈底当成了基准，心想：“在基准之上为正，在基准之下当然为负了”，其实不然，栈的段界限是以栈段的基址为基准的，并不是以栈底，因此栈的段界限肯定是位于栈顶之下。地址本身由低向高发展，段界限也是个地址，而栈的扩展方向是由高地址向低地址，与段界限有个碰撞的趋

势。为了避免碰撞，将段界限地址+1 视为栈可以访问的下限。段界限+1，才是栈指针可达的下边界，如图 4-18 所示。



▲图 4-18　栈的段界限

下面咱们以 32 位保护模式下的栈为例简要说下。

32 位保护模式下栈的栈顶指针是 esp 寄存器，栈的操作数大小是由 B 位决定的，我们这里假设 B 为 1，即操作数是 32 位。栈段也是位于内存中，所以它也要受控于段描述符中的 G 位。

- 如果 G 为 0，实际的段界限大小=描述符中的段界限。
- 如果 G 为 1，实际的段界限大小=描述符中段界限*0x1000+0xFFF。

同代码段的操作数一样，用于压栈的操作数也有其长度，push 指令每向栈中压入操作数时，实际上就是将 esp 指针减去操作数的大小（2 字节或 4 字节）后，再将操作数复制到 esp 减 4 后的新地址。栈指针可访问的最低地址是由实际段界限决定的，但栈段最大可访问的地址是由 B 位决定的，我们这里 B 位为 1，表示 32 位操作数，所以栈指针最大可访问地址是 0xFFFFFFFF。综上所述，每次向栈中压入数据时就是 CPU 检查栈段的时机，它要求必须满足以下条件。

实际段界限+1≤esp-操作数大小≤ 0xFFFFFFFF

假设现在 esp 指针为 0xFFFFE002，段描述符的 G 位为 1，描述符中的段界限为 0xFFFFD。故实际段界限为 0x1000*FFFFD+0xFFF=0xFFFFDFFF。当执行 push ax，压入 2 字节的操作数，即 esp-2=0xFFFFE000，新的 esp 值≥实际段界限 0xFFFFDFFF +1。如果执行 push eax，压入 4 字节的数据，esp-4=0xFFFFDFFE，小于实际段界限 0xFFFFDFFF，故 CPU 会抛出异常。

由于 esp 只是栈段内的偏移地址，其真正物理地址还要加上段基址。假设段基址为 0，故该栈段：

最大可访问地址为 0+0xFFFFFFFF=0xFFFFFFFF。

最小可访问地址为 0+0xFFFFDFFF+1=0xFFFFE000。

栈段空间大小为 0xFFFFFFFF-0xFFFFE000=8KB。

由于本书中并不会用到向下扩展的数据段，故有关于此的内容到这里就结束了，咱们把更多的精力放在后面的操作系统上。

本章到此结束，下章咱们再见。

第 5 章　保护模式进阶，向内核迈进

在上一章中虽然介绍了保护模式，但过多的笔墨是花在了理论上，对于实践只是浅尝辄止。本章之前的所有章节属于偏理论基础，毕竟有了扎实的基本功才能消化更深入的知识。

从本章开始，我们的代码将在保护模式下工作，除了开启虚拟内存外，我们还会接触到其他硬件，本章献给大家的全是“硬”货。从这一刻起，我们才算开始了真正的操作系统学习之旅。

5.1 获取物理内存容量

操作系统是计算机硬件的管家，它不仅要知道自己安装了哪些硬件，还得给出有效得当的管理措施，按照预定的一套管理策略使硬件资源得到合理的运用。但管理策略只是逻辑上的东西，是操作系统自圆其说的一套管理资源的方法，管理再漂亮，没有硬件支撑也无能为力，真正干活的都是底层。

保护模式最“大”的特点就是寻址空间“大”，在进入保护模式之后，我们将接触到虚拟内存、内存管理等。但这些和内存有关的概念都建立在物理内存之上，无论理论概念说得多高大上，最终也要在物理内存上落实行动。为了在后期做好内存管理工作，咱们先得知道自己有多少物理内存才行。

5.1.1 学习 Linux 获取内存的方法

在 Linux 中有多种方法获取内存容量，如果一种方法失败，就会试用其他方法。比如在 Linux 2.6 内核中，是用 detect_memory 函数来获取内存容量的。其函数在本质上是通过调用 BIOS 中断 0x15 实现的，分别是 BIOS 中断 0x15 的 3 个子功能，子功能号要存放到寄存器 EAX 或 AX 中，如下。

- EAX=0xE820：遍历主机上全部内存。
- AX=0xE801：分别检测低 15MB 和 16MB～4GB 的内存，最大支持 4GB。
- AH=0x88：最多检测出 64MB 内存，实际内存超过此容量也按照 64MB 返回。

BIOS 中断是实模式下的方法，只能在进入保护模式前调用。咱们效仿 Linux“不弃不舍”的精神，在实模式下也用这三种方法检测完内存容量后再进入保护模式。如果一种方法获取失败，尝试另一种方法，若三种方法都失败了，由于无法获取内存信息，后续程序无法加载，只好将机器挂起，停止运行。

BIOS 中断可以返回已安装的硬件信息，由于 BIOS 及其中断也只是一组软件，它要访问硬件也要依靠硬件提供的接口，所以，获取内存信息，其内部是通过连续调用硬件的应用程序接口（Application Program Interface，API）来获取内存信息的。另外，由于每次调用 BIOS 中断都是有一定的代价的（比如至少要将程序的上下文保护起来以便从中断返回时可以回到原点继续向下执行），所以尽量在一次中断中返回足量的信息，由用户程序自己挑出重点内容。下面介绍的中断便是这方面的典范。

BIOS 0x15 中断提供了丰富的功能，具体要调用的功能，需要在寄存器 ax 中指定。其中 0xE8xx 系列的子功能较为强大，0x15 中断的子功能 0xE820 和 0xE801 都可以用来获取内存，区别是 0xE820 返回的是内存布局，信息量相对多一些，操作也相对复杂。而 0xE801 直接返回的是内存容量，操作适中，不繁不简。上面还提到了子功能 0x88 也能获取内存容量，这是最简单的用法，不过操作越简单，功能也就越薄弱。话说 Linux 中的 dmesg 命令就与 0xE820 相关，可见其功能是很大的，咱们就按照功能强弱的顺序逐一介绍用法。

5.1.2 利用 BIOS 中断 0x15 子功能 0xe820 获取内存

咱们先介绍 0xE820 子功能，这是最灵活的内存获取方式。

BIOS 中断 0x15 的子功能 0xE820 能够获取系统的内存布局，由于系统内存各部分的类型属性不同，BIOS 就按照类型属性来划分这片系统内存，所以这种查询呈迭代式，每次 BIOS 只返回一种类型的内存信息，直到将所有内存类型返回完毕。子功能 0xE820 的强大之处是返回的内存信息较丰富，包括多个属性字段，所以需要一种格式结构来组织这些数据。内存信息的内容是用地址范围描述符来描述的，用于存储这种描述符的结构称之为地址范围描述符（Address Range Descriptor Structure，ARDS），格式见表 5-1。

表 5-1　地址范围描述符结构 ARDS

字节偏移量	属性名称	描　述
0	BaseAddrLow	基地址的低 32 位
4	BaseAddrHigh	基地址的高 32 位
8	LengthLow	内存长度的低 32 位，以字节为单位
12	LengthHigh	内存长度的高 32 位，以字节为单位
16	Type	本段内存的类型

此结构中的字段大小都是 4 字节，共 5 个字段，所以此结构大小为 20 字节。每次 int 0x15 之后，BIOS 就返回这样一个结构的数据。注意，ARDS 结构中用 64 位宽度的属性来描述这段内存基地址（起始地址）及其长度，所以表中的基地址和长度都分为低 32 位和高 32 位两部分。

其中的 Type 字段用来描述这段内存的类型，这里所谓的类型是说明这段内存的用途，即其是可以被操作系统使用，还是保留起来不能用。Type 字段的具体意义见表 5-2。

表 5-2　地址范围描述符结构的 Type 字段

Type 值	名　称	描　述
1	AddressRangeMemory	这段内存可以被操作系统使用
2	AddressRangeReserved	内存使用中或者被系统保留，操作系统不可以用此内存
其他	未定义	未定义，将来会用到，目前保留。但是需要操作系统一样将其视为 ARR（AddressRangeReserved）

为什么 BIOS 会按类型来返回内存信息呢？原因是这段内存可能是。

- 系统的 ROM。
- ROM 用到了这部分内存。
- 设备内存映射到了这部分内存。
- 由于某种原因，这段内存不适合标准设备使用。

由于我们在 32 位环境下工作，所以在 ARDS 结构属性中，我们只用到低 32 位属性。BaseAddrLow+LengthLow 是一片内存区域上限，单位是字节。正常情况下，不会出现较大的内存区域不可用的情况，除非安装的物理内存极其小。这意味着，在所有返回的 ARDS 结构里，此值最大的内存块一定是操作系统可使用的部分，即主板上配置的物理内存容量。

BIOS 中断只是一段函数例程，调用它就要为其提供参数，现在介绍下 BIOS 中断 0x15 的 0xe820 子功能需要哪些参数。

先介绍下此中断例程的调用方法。表 5-3 所示是使用此中断的方法，分输入和输出两部分。

表 5-3　BIOS 中断 0x15 子功能 0xE820 说明

调用或返回	寄存器或状态位	参 数 用 途
调用前输入	EAX	子功能号：EAX 寄存器用来指定子功能号，此处输入为 0xE820
	EBX	ARDS 后续值：内存信息需要按类型分多次返回，由于每次执行一次中断都只返回一种类型内存的 ARDS 结构，所以要记录下一个待返回的内存 ARDS，在下一次中断调用时通过此值告诉 BIOS 该返回哪个 ARDS，这就是后续值的作用。第一次调用时一定要置为 0，EBX 具体值我们不用关注，字取决于具体 BIOS 的实现。每次中断返回后，BIOS 会更新此值
	ES：DI	ARDS 缓冲区：BIOS 将获取到的内存信息写入此寄存器指向的内存，每次都以 ARDS 格式返回

续表

调用或返回	寄存器或状态位	参 数 用 途
调用前输入	ECX	ARDS 结构的字节大小：用来指示 BIOS 写入的字节数。调用者和 BIOS 都同时支持的大小是 20 字节，将来也许会扩展此结构
	EDX	固定为签名标记 0x534d4150，此十六进制数字是字符串 SMAP 的 ASCII 码：BIOS 将调用者正在请求的内存信息写入 ES：DI 寄存器所指向的 ARDS 缓冲区后，再用此签名校验其中的信息
返回后输出	CF 位	若 CF 位为 0 表示调用未出错，CF 为 1，表示调用出错
	EAX	字符串 SMAP 的 ASCII 码 0x534d4150
	ES:DI	ARDS 缓冲区地址，同输入值是一样的，返回时此结构中已经被 BIOS 填充了内存信息
	ECX	BIOS 写入到 ES:DI 所指向的 ARDS 结构中的字节数，BIOS 最小写入 20 字节
	EBX	后续值：下一个 ARDS 的位置。每次中断返回后，BIOS 会更新此值，BIOS 通过此值可以找到下一个待返回的 ARDS 结构，咱们不需要改变 EBX 的值，下一次中断调用时还会用到它。在 CF 位为 0 的情况下，若返回后的 EBX 值为 0，表示这是最后一个 ARDS 结构

表中的 ECX 寄存器和 ES：DI 寄存器，是典型的“值-结果”型参数，即调用方提供了两个变量作为被调用函数的参数，一个变量是缓冲区指针，另一个变量是缓冲区大小。被调用函数在缓冲区中写入数据后，将实际所写入的字节数记录到缓冲区大小变量中。

根据表 5-3 中的说明，此中断的调用步骤如下。

（1）填写好“调用前输入”中列出的寄存器。

（2）执行中断调用 int 0x15。

（3）在 CF 位为 0 的情况下，“返回后输出”中对应的寄存器便会有对应的结果。

5.1.3 利用 BIOS 中断 0x15 子功能 0xe801 获取内存

另一个获取内存容量的方法是 BIOS0x15 中断的子功能 0xE801。

此方法虽然简单，但功能也不强大，最大只能识别 4GB 内存，不过这对咱们 32 位地址总线足够了。稍微有点不便的是此方法检测到的内存是分别存放到两组寄存器中的。低于 15MB 的内存以 1KB 为单位大小来记录，单位数量在寄存器 AX 和 CX 中记录，其中 AX 和 CX 的值是一样的，所以在 15MB 空间以下的实际内存容量=AX*1024。AX、CX 最大值为 0x3c00，即 0x3c00*1024=15MB。16MB～4GB 是以 64KB 为单位大小来记录的，单位数量在寄存器 BX 和 DX 中记录，其中 BX 和 DX 的值是一样的，所以 16MB 以上空间的内存实际大小=BX*64*1024，不用在意 BX 和 DX 最大值是多少，前面说过啦，只支持 4GB 空间，您可以反推一下看看。

咱们还是列个表，将其用法分为输入、输出两部分介绍。表 5-4 所示是子功能 0xE801 的使用方法，较表 5-3 确实容易不少。

表 5-4 BIOS 中断 0x15 子功能 0xE801 说明

调用或返回	寄存器或状态位	用途	描述
调用前输入	AX	Function Code	子功能号：0xE801
返回后输出	CF 位	Carry Flag	若 CF 位为 0 表示调用未出错，CF 为 1，表示调用出错
	AX	Extended 1	以 1KB 为单位，只显示 15MB 以下的内存容量，故最大值为 0x3c00，即 AX 表示的最大内存为 0x3c00*1024=15MB
	BX	Extended 2	以 64KB 为单位，内存空间 16MB～4GB 中连续的单位数量，即内存大小为 BX*64*1024 字节
	CX	Configured 1	同 AX
	DX	Configured 2	同 BX

和表 5-3 相比，表 5-4 多了“用途”列，而且是英文，一会就知道用在哪里了。再次提醒，中断返回后，AX 和 CX 中，其值的单位是 1KB，而 BX 和 DX 的单位是 64KB。

当初我看到这儿的时候，脑子中不禁弹出了两个问号。

（1）为什么要分“前 15MB”和“16MB 以上”这两部分来展示 4GB 内存？

（2）为什么寄存器结果是重复的？如寄存器 AX 和 CX 相等，BX 和 DX 相等？

为了解决第 1 个问题，让我们实际测试下，让事实说话。测试方法是修改 bochs 配置文件 bochsrc.disk 中的内存容量参数 megs，然后执行 BIOS 中断。测试结果见表 5-5。

表 5-5　　BIOS 中断 0x15 子功能 0xE801 实例

实际物理内存	AX	BX	检测到的内存大小
14MB	0x3400	0	AX*1024+BX*64*1024=13MB
15MB	0x3800	0	AX*1024+BX*64*1024=14MB
16MB	0x3C00	0	AX*1024+BX*64*1024=15MB
17MB	0x3C00	0x10	AX*1024+BX*64*1024=16MB
18MB	0x3C00	0x20	AX*1024+BX*64*1024=17MB

表 5-5 中“实际物理内存”和“检测到的内存大小”，它们之间总是差 1MB，言外之意是总有 1MB 内存不可用。这是怎么回事？真是一波未平一波又起啊。

很多问题都是祖上传下来的，即著名的历史遗留问题。80286 拥有 24 位地址线，其寻址空间是 16MB。当时有一些 ISA 设备要用到地址 15MB 以上的内存作为缓冲区，也就是此缓冲区为 1MB 大小，所以硬件系统就把这部分内存保留下来，操作系统不可以用此段内存空间。保留的这部分内存区域就像不可以访问的黑洞，这就成了内存空洞 memory hole。现在虽然很少很少能碰到这些老 ISA 设备了，但为了兼容，这部分空间还是保留下来，只不过是通过 BIOS 选项的方式由用户自己选择是否开启。BIOS 厂商不同，一般的菜单选项名称也不相同，不过大概意思都差不多。比如咱们开机进入 BIOS 界面后，会有类似这样的选项：

memory hole at address 15m-16m

将此选项设为 enable 或 disable 便开启或关闭对这类扩展 ISA 设备的支持。

话说，起初定义这个 0xe801 子功能，就是为了支持扩展 ISA 服务。现在来回答这个问题。

如果检测到的内存容量大于等于 16MB，BIOS 0x15 中断返回的结果中，AX*1024 必然是小于等于 15MB，而 BX*64*1024 肯定大于 0。所以，内存容量分成两部分展示，只要符合这两个结果，就能检查出内存空洞。当然如果物理内存在 16MB 以下，此方法就不灵了，但检测到的内存依然会小于实际内存 1MB。所以实际的物理内存大小，在检测结果的基础上一定要加上 1MB。

至于第 2 个疑问，手册上是这么说的：

Not sure what this difference between the "Extended" and "Configured" numbers are, but they appear to be identical, as reported from the BIOS.

这句英文中的两个单词"Extended"和"Configured"已经在表 5-4 的“用途”列中出现了，后面数字相同的为一组，比如 AX 的用途为 Extended 1，CX 的用途为 Configured 1，AX 和 CX 为一组，BX 和 DX 类同。

这句英文大概意思是：不清楚"Extended" 和 "Configured"之间的区别，但它们似乎是相同的，BIOS 就是这样说的。咱们这里暂时就不深究了，毕竟咱们只是想拿到内存容量，以后等咱们有精力了再深入学习吧。

此中断的调用步骤如下。

（1）将 AX 寄存器写入 0xE801。

（2）执行中断调用 int 0x15。

（3）在 CF 位为 0 的情况下，“返回后输出”中对应的寄存器便会有对应的结果。

5.1.4　利用 BIOS 中断 0x15 子功能 0x88 获取内存

最后一个获取内存的方法也同样是 BIOS 0x15 中断，子功能号是 0x88。该方法使用最简单，但功能也最简单，简单到只能识别最大 64MB 的内存。即使内存容量大于 64MB，也只会显示 63MB，大家可以自己在 bochs 中试验下。为什么只显示到 63MB 呢？因为此中断只会显示 1MB 之上的内存，不包括这 1MB，

咱们在使用的时候记得加上 1MB。参数见表 5-6。

表 5-6　　BIOS 中断 0x15 子功能 0x88 说明

调用或返回	寄存器或状态位	参数用途
调用前输入	AH	子功能号：0x88
返回后输出	CF 位	若 CF 位为 0 表示调用未出错，CF 为 1，表示调用出错
	AX	以 1KB 为单位大小，内存空间 1MB 之上的连续单位数量，不包括低端 1MB 内存。故内存大小为 AX*1024 字节+1MB

中断返回后，AX 寄存器中的值，其单位是 1KB。此中断的调用步骤如下。

（1）将 AX 寄存器写入 0x88。

（2）执行中断调用 int 0x15。

（3）在 CF 位为 0 的情况下，“返回后输出”中对应的寄存器便会有对应的结果。

5.1.5　实战内存容量检测

以上介绍了三种检测内存的方法，是时候将它们一网打尽测试一把啦。

不知道各位怎么想的，反正我已经迫不及待地想上机器上测试一下啦，上菜，见代码 5-1。

代码 5-1　（project/c5/a/boot/loader.S）

```
 1   %include "boot.inc"
 2   section loader vstart=LOADER_BASE_ADDR
 3   LOADER_STACK_TOP equ LOADER_BASE_ADDR
 4
 5 ;构建 gdt 及其内部的描述符
…略
此处是定义 GDT，未有新变化
…略
20    times 60 dq 0
…略
 25    ; total_mem_bytes 用于保存内存容量，以字节为单位，此位置比较好记
 26    ; 当前偏移 loader.bin 文件头 0x200 字节
       ;  loader.bin 的加载地址是 0x900
 27    ; 故 total_mem_bytes 内存中的地址是 0xb00
       ; 将来在内核中咱们会引用此地址
 28    total_mem_bytes dd 0
 29    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 30
 31;以下是定义 gdt 的指针，前 2 字节是 gdt 界限，后 4 字节是 gdt 起始地址
 32    gdt_ptr  dw  GDT_LIMIT
 33             dd GDT_BASE
 34
 35;人工对齐:total_mem_bytes4+gdt_ptr6+ards_buf244+ards_nr2，共 256 字节
 36    ards_buf times 244 db 0
 37    ards_nr dw 0                   ;用于记录 ARDS 结构体数量
38
39    loader_start:
 40
 41 ; int 15h eax = 0000E820h ,edx = 534D4150h ('SMAP') 获取内存布局
42
 43      xor ebx, ebx                 ;第一次调用时，ebx 值要为 0
 44      mov edx, 0x534d4150          ;edx 只赋值一次，循环体中不会改变
 45      mov di, ards_buf             ;ards 结构缓冲区
 46 .e820_mem_get_loop:               ;循环获取每个 ARDS 内存范围描述结构
 47      mov eax,  0x0000e820         ;执行 int 0x15 后，eax 值变为 0x534d4150，
;所以每次执行 int 前都要更新为子功能号
 48      mov ecx,  20                 ;ARDS 地址范围描述符结构大小是 20 字节
49      int 0x15
 50      jc .e820_failed_so_try_e801
;若 cf 位为 1 则有错误发生，尝试 0xe801 子功能
 51      add di, cx                   ;使 di 增加 20 字节指向缓冲区中新的 ARDS 结构位置
 52      inc word [ards_nr]           ;记录 ARDS 数量
53     cmp ebx, 0                     ;若 ebx 为 0 且 cf 不为 1，这说明 ards 全部返回
                 ; 当前已是最后一个
```

```
 54      jnz .e820_mem_get_loop
 55
 56 ;在所有 ards 结构中
   ;找出(base_add_low + length_low)的最大值，即内存的容量
 57      mov cx, [ards_nr]
;遍历每一个 ARDS 结构体,循环次数是 ARDS 的数量
 58      mov ebx, ards_buf
 59      xor edx, edx                  ;edx 为最大的内存容量，在此先清 0
 60 .find_max_mem_area:
;无需判断 type 是否为 1,最大的内存块一定是可被使用的
 61      mov eax, [ebx]                ;base_add_low
 62      add eax, [ebx+8]              ;length_low
63     add ebx, 20                     ;指向缓冲区中下一个 ARDS 结构
 64      cmp edx, eax
;冒泡排序,找出最大,edx 寄存器始终是最大的内存容量
 65      jge .next_ards
 66      mov edx， eax                 ;edx 为总内存大小
 67 .next_ards：
 68     loop .find_max_mem_area
 69     jmp .mem_get_ok
70
 71 ;------  int 15h ax = E801h 获取内存大小，最大支持 4G  ------
 72 ; 返回后, ax cx 值一样,以 KB 为单位，bx dx 值一样，以 64KB 为单位
 73 ; 在 ax 和 cx 寄存器中为低 16MB，在 bx 和 dx 寄存器中为 16MB 到 4GB
 74 .e820_failed_so_try_e801：
 75      mov ax,0xe801
 76      int 0x15
 77      jc .e801_failed_so_try88      ;若当前 e801 方法失败，就尝试 0x88 方法
 78
 79 ;1 先算出低 15MB 的内存
   ;  ax 和 cx 中是以 KB 为单位的内存数量，将其转换为以 byte 为单位
 80      mov cx,0x400                  ;cx 和 ax 值一样，cx 用作乘数
 81      mul cx
 82      shl edx,16
 83      and eax,0x0000FFFF
 84      or edx,eax
 85      add edx, 0x100000             ;ax 只是 15MB，故要加 1MB
 86      mov esi,edx                   ;先把低 15MB 的内存容量存入 esi 寄存器备份
 87
 88 ;2 再将 16MB 以上的内存转换为 byte 为单位
   ;  寄存器 bx 和 dx 中是以 64KB 为单位的内存数量
 89      xor eax，eax
 90      mov ax，bx
 91      mov ecx， 0x10000             ;0x10000 十进制为 64KB
 92      mul ecx                       ;32 位乘法，默认的被乘数是 eax，积为 64 位
                                       ;高 32 位存入 edx，低 32 位存入 eax
 93      add esi，eax
;由于此方法只能测出 4GB 以内的内存，故 32 位 eax 足够了
        ; edx 肯定为 0，只加 eax 便可
 94      mov edx，esi                  ;edx 为总内存大小
 95      jmp .mem_get_ok
 96
 97 ;-----  int 15h ah = 0x88 获取内存大小，只能获取 64MB 之内  -----
 98 .e801_failed_so_try88:
 99      ;int 15 后，ax 存入的是以 KB 为单位的内存容量
100      mov  ah, 0x88
101      int  0x15
102      jc .error_hlt
103      and eax,0x0000FFFF
104
105     ;16 位乘法，被乘数是 ax，积为 32 位。积的高 16 位在 dx 中
;积的低 16 位在 ax 中
106      mov cx, 0x400
;0x400 等于 1024,将 ax 中的内存容量换为以 byte 为单位
107      mul cx
108      shl edx, 16                   ;把 dx 移到高 16 位
109      or edx, eax                   ;把积的低 16 位组合到 edx，为 32 位的积
110      add edx,0x100000              ;0x88 子功能只会返回 1MB 以上的内存
                          ;故实际内存大小要加上 1MB
111
112 .mem_get_ok：
113     mov [total_mem_bytes]， edx
```

```
;将内存换为 byte 单位后存入 total_mem_bytes 处
…略
```

代码 5-1 是为了演示 BIOS 中断 0x15 的用法，咱们把以上介绍的三种获取内存的方法都用上了。代码中的注释还是很全的，再结合前面介绍的用法，大家应该很容易看懂。不过我已经厚道惯了，多多少少还是要给大家说一下，代码虽然非常简单，但毕竟揣摩别人的思想还是要花时间的。

本代码第 28 行定义了 4 字节的变量 total_mem_bytes，此变量用于存储获取到的内存容量，以字节为单位。也是啊，经过千辛万苦才获取到的内存大小当然要赶紧找个地方藏起来了，哈哈，有点夸张了，总之先存起来留着以后用。

在 total_mem_bytes 定义的地方上面有几行注释，阐述了 total_mem_bytes 的地址是 0xb00。理由是它前面有 4 个段描述符的定义，还有预留 60 个段描述槽位 times 60 dq 0。段描述符大小是 8 字节，dq 也是 8 字节，所以偏移量是 (4+60)*8=512=0x200 字节。本程序的加载地址是 0x900，0x900+0x200=0xb00，所以 0xb00 便是变量 total_mem_bytes 加载到内存中的地址。将来在内核中实现内存分配系统时还会用到此地址。

第 35～37 行是提前定义的缓冲区，为的是存储 BIOS 0x15 中断 0xe820 子功能返回的 ARDS 结构。每执行该中断一次便会得到一个 ARDS 结构的数据，按理说缓冲区的大小等于 ARDS 结构的大小就行了，但为了编程省事，处理的思路是将所有 ARDS 都得到后再统一遍历，所以我们就申请个大一些的缓冲区。缓冲区地址是 ards_buf，但缓冲区大小是多少合适呢？已知一个 ARDS 结构是 20 字节，具体多大取决于到底会有多少个 ARDS 结构。所以先估计个数吧，不够了再加（在本机实际测试中共返回了 6 个 ARDS 结构）。在这里先为其分配 244 字节，哈哈，有点奇怪是不，为什么有零有整的？这是为了手工对齐下面 39 行的标签 loader_start，使其在文件内的偏移地址为 0x300。这纯属个人喜好，就是想凑个整数，显得好看。您看，total_mem_bytes 是 4 字节，gdt_ptr 是 6 字节，ards_buf 是 244 字节，ards_nr 是 2 字节，加起来的和是 256 字节，即 0x100。加上 total_mem_bytes 在文件内偏移地址是 0x200，所以 loader_start 在文件内的偏移地址是 0x100+0x200=0x300。代码“ards_buf times”的大小是 244 字节纯粹是凑出来的，无实际意义，哈哈，这属于传说中的强迫症行为。

上一版本 loader.S（project/c4/a/boot/loader.S）中第 4 行的代码是 jmp loader_start，上一版本 mbr.S（project/c4/a/boot/mbr.S）的第 55 行是跳转指令 jmp LOADER_BASE_ADDR，经过这两个跳转才执行到 loader_start 处的代码。在本节版本中，loader_start 是 loader.S 中第一条指令的起始地址，那个跳转指令没有了。所以，在本节主引导记录程序 mbr.S 中的跳转指令已经变成了 jmp LOADER_BASE_ADDR + 0x300，多加了个 0x300 字节，跨过前面的数据部分，直接跳到 loader_start。之所以这样做，是为对齐代码，因为在程序开头加个跳转指令，其机器码要占用 3 字节空间，原本在它之后定义的数据，其地址未对齐到偶数，这会影响硬件执行的速度。而且，将来在内核中引用 total_mem_bytes 的地址时也要用个奇数，感觉很别扭。

从代码的第 41 行开始，采用 BIOS 中断 0x15 的三种子功能来检测内存。在检测内存时，必然是先使用功能最全、检测功能最强大的方法，功能最弱的、检测能力有限的方法应该是在“万般无奈”下才用，要放在最后，以避免无法检测出真实的内存容量。从强到弱的子方法依次是 0xe820、0xe801、0x88。

第 41～69 行用的子功能 0xe820 方法。此方法需要提前准备好一块数据缓冲区用于存放返回的 ARDS 结构，此缓冲区我们在上面已经准备好了，就是 ards_buf。按照 0xe820 的调用方法，es：di 存放缓冲区地址，由于 es 在 mbr 中已经赋值了，所以在第 47 行“mov di, ards_buf”，只为 di 赋值便可。

每执行一次 int 0x15 后，寄存器 eax、ebx、ecx 都会更新。eax 的值由之前的子功能号变成了字符串 SMAP 的 ASCII 码，ebx 为新的后续值，ecx 为实际写入缓冲区中的字节数。其中 ebx 咱们不用干涉，原方不动地作为输入即可。eax 和 ecx 寄存器每次调用前都要更新为正确的输入参数，所以放在了循环体中。接下来每得到一个 ARDS 结构后，便将 di 增加一个 ARDS 结构大小（这里是 20 字节），以指向缓冲区中的下一个 ARDS 存放的位置，然后将变量 ards_nr 加 1，以记录 ARDS 的个数，用于在后面的代码中遍历所有 ARDS，找出最大内存块。

第 56～69 行是找出最大的内存块。思路是对每一个 ARDS 结构中的 BaseAddrLow 与 LengthLow 相加求和，遍历完所有 ARDS，值最大的则为内存容量，由于 BaseAddrLow+LengthLow 的单位是字节而无需转换，之后便直接跳转到.mem_get_ok，将此容量数写入变量 total_mem_bytes，具体代码为第 113 行的 mov [total_mem_bytes], edx。在此说明下，三种方法探测到的内存容量都是统一跳转到.mem_get_ok 处后以字节形式写入到变量

total_mem_bytes，所以三种方法中内存容量都要用 edx 来保存。

第 71～95 行是利用子功能 0xe801 探测内存容量。由于方法本身比较简单，所以代码量很短。在 76 行执行中断后，第 79～86 行先计算出低 15MB 内存空间的容量。这里面用到了乘法指令 mul，在 16 位乘法中，由于 mul 指令固定的乘数是寄存器 AX，所以只给提供另一个乘数就行了，于是乘法指令格式是 mul 16 位内存或 16 位寄存器。结果（积）是 32 位，高 16 在 DX 寄存器，低 16 位在 AX 寄存器。

由于寄存器 AX 和 CX 的单位是 KB，这里要将获得的结果转换成字节，所以寄存器 AX 或 CX 要乘以 1024。前面说过啦，寄存器 AX 和 CX 值相同，随便用哪个做乘数皆可，由于固定的操作数 AX，所以把 CX 的值覆盖为 0x400，即 1024。这里用的是 16 位操作数乘法，积的高 16 位在 DX 寄存器，低 16 位在 AX 寄存器。所以将 EDX 左移 16 位后再与 AX 做或运算便得到了完整 32 位的积。在第 85 行将 edx 加了 1MB，原因是获取到的内存总比实际大小少 1MB，故在此“补偿”。后面的乘法指令会破坏寄存器 EDX 的值，所以第 86 行将结果备份到寄存器 ESI。

第 88～95 行是计算 16MB 之上的内存容量，结果存放在寄存器 BX 和 DX，单位是 64KB，所以也要将其转换为字节。这里面用到了 32 位操作数的乘法，32 位乘法固定的乘数是 EAX，同样也是再提供一个操作数就行了。所以其指令格式为 mul 32 位内存或 32 位寄存器。积为 64 位结果，高 32 位在 EDX 寄存器，低 32 位在 EAX 寄存器。第 90 行用寄存器 BX 初始化 EAX 寄存器的低 16 位 AX（EAX 的高 16 位已经由 xor eax，eax 清 0），EAX 为固定的乘数，另一个乘数是 0x10000，即 64KB。由于 0xe801 子功能只能测出 4GB 之内的内存容量，所以只需要乘积的低 32 位结果，也就是寄存器 EAX 的值就够了，最后再将备份在寄存器 ESI 中的低 15MB 空间的内存容量同 EAX 相加，存入变量 total_mem_bytes 中。

第 98～110 行是用子功能 0x88 方法探测内存容量，功能简单，代码更简单，实际代码同上个方法类似，容易看懂，不多说了。

方法说完了，该是实际运行检测的时候了。虽然内存容量检测出来了，但代码中并没有将它显示出来，原因是我们将很快实现打印函数了，这里就不先“凑合”打印了。不过我们依然能够在 bochs 中检验结果，大家肯定都想到了，对，就是通过调试手段来查看变量 total_mem_bytes 的内存就行啦。事不宜迟，我们看一下实际运行情况吧。

检测前得先知道机器上到底装了多少内存才行，否则谁知道检测的结果是否正确。幸亏咱们是在虚拟机上，否则还得拆机箱拔内存条，当然是开玩笑了，还是有其他软件方法能够测试出来的。咱们只需要查看 bochs 虚拟机的配置文件 bochsrc.disk 就行了，其中的 megs 参数用来指定内存大小，如图 5-1 所示。

```
[work@localhost bochs]$ cat -n bochsrc.disk
     1  ###############################################
     2  # Configuration file for Bochs
     3  ###############################################
     4
     5  # 第一步，首先设置Bochs在运行过程中能够使用的内存，本例为32MB。
     6  # 关键字为：megs
     7
     8  megs: 32
     9
```

▲图 5-1　虚拟机内存容量

由图 5-1 可见，机器上装了 32MB 内存。megs 指定的内存以 MB 为单位，只要给出数值部分就好啦。

激动人心的时候到了，bochs 走起。好久不开 bochs 了，跟大伙儿一块复习下。到 bochs 安装目录下执行。

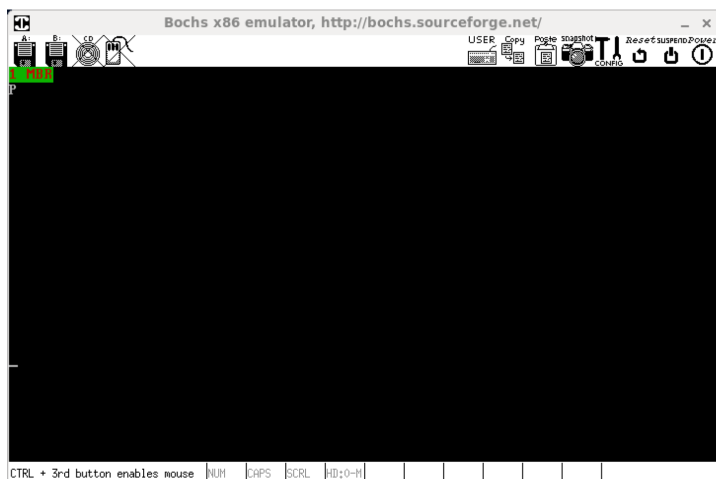
（1）bin/bochs –f bochsrc.disk 回车。

（2）弹出提示菜单：Please choose one: [6]，这说明默认是选项 6，回车。

（3）在 bochs 控制台中键入命令 c 回车。

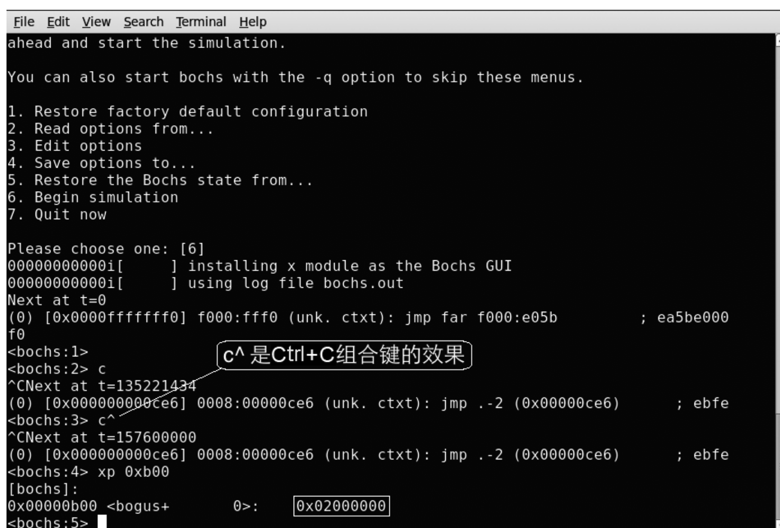
之后程序就跑起来了，在虚拟机中的运行效果如图 5-2 所示。

这张图不但没有帮助，似乎还帮了“倒忙”，就连之前的屏幕左下角的字符串“2 loader in real.”都没了。贴这张图的目的就是想告诉大伙儿，代码改了，打印这段字符的 BIOS 0x10 中断“下线”啦。因为以后在保护模式中经常显示这段文字，我怕误导大家。



▲图 5-2　bochs 调试检测

前面已经说过啦，变量 total_mem_bytes 的地址是 0xb00，所以咱们在 bochs 控制台中用 xp 指令来查看该地址就行了，说干就干，先用 Ctrl+C 组合键中断 bochs 的运行，再输入命令 xp 0xb00 回车，欲知详情见图 5-3。



▲图 5-3　用 xp 命令查看物理内存

如图 5-3 所示，在执行 xp 0xb00 后，结果是 0x02000000，图中已经用框框标出来了。0x2000000 换算成十进制正是 32MB，可见检测结果是正确的。大家用计算器转换一下便知道了。如果大家不嫌弃，小弟这里提供个进制转换脚本 calculator.sh，为了方便大家，我将其放在随书代码的 tool 目录中。

代码很简单，就一行，用的是 awk 的 printf 函数。

```
echo |awk "{printf(\"%$2\n\",$1)}"
```

使用方法也很简单，第 1 个参数是待转换的数字，一般进制都支持，第 2 个参数是输出的格式。用法和 C 语言的 printf 一样。

例如：sh calculator.sh 0x02000000/1024/1024 f 回车。

32.000000

32.000000 是结果。用浮点数格式 f 来显示结果的好处是避免用整数时的取整，误以为整除了。

虽然我们介绍了三个检测方法，但这个 0x02000000 却是 BIOS 0x15 中断子功能 0xe820 的功劳，另外两个子功能没用上。大家可以在 bochs 中跟一下代码执行的情况，本程序实际运行情况是在方法 0xe820 中，int 0x15 执行了 6 次，返回了 6 个 ARDS 结构。

说了这么久，内存检测部分终于完成了，对物理内存做到心中有数后，下面咱们该开始步入虚拟内存啦。

5.2 启用内存分页机制，畅游虚拟空间

在此之前，我们的程序虽然已经进了保护模式，地址空间到达了前所未有的 4GB，但其依然是受限制的，就像共享网络带宽一样，此进程要和其他进程包括操作系统共享这 4GB 内存空间。我们把段基址+段内偏移地址称为线性地址，线性地址是唯一的，只能属于某一个进程。

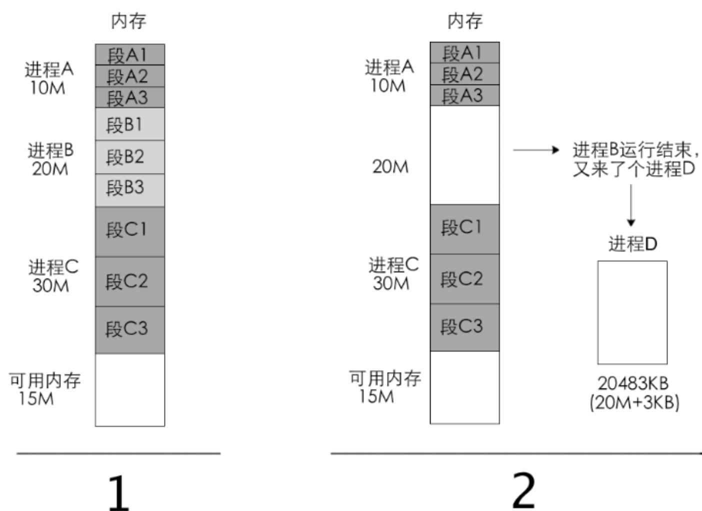
在我们的机器上即使只有 512MB 的内存，每个进程自己的内存空间也是 4GB，不用说大家也都知道这是指虚拟内存空间。我想大多数同学也仅仅是知道虚拟地址的概念，未必知道虚拟地址的实现原理。就像大家都知道电脑开机必须要接上电源才行，但不是所有人都知道电是怎样将整个硬件系统发动起来的一样。本节就要将这层神秘面纱揭开，跟大家说说这个神秘面纱后面的那些事儿。

5.2.1 内存为什么要分页

一直以来我们都直接在内存分段机制下工作，目前未出问题看似良好，的确，目前咱们的应用过于简单了，就一个 loader 在跑，能出什么问题呢？可是想象一下，当我们物理内存不足时会怎么办呢？比如系统里的应用程序过多，或者内存碎片过多无法容纳新的进程，或者曾经被换出到硬盘中的内存段需要再次重新装到内存，可是内存中找不到合适大小的内存区域怎么办？也许有人会说，这简单啊，停止想象呗……嘿嘿，开玩笑而已，问题还是要解决的。

也许文字说得并不是很清楚，下面以图示说明这些情况。

图 5-4 所示模拟了多个进程并行的情况。在第 1 步中，系统里有 3 个进程正在运行，进程 A、B、C 各占了 10MB、20MB、30MB 的内存空间，物理内存还是挺宽裕的，还剩下 15MB 可用。到了第 2 步就悲催了，此时进程 B 已经运行结束，腾出了 20MB 的内存，可是待加载运行的进程 D 需要 20MB+3KB 的内存空间，即 20483KB。现在的运行环境未开启分页功能，“段基址+段内偏移”产生的线性地址就是物理地址，程序中引用的线性地址是连续的，所以物理地址也连续。虽然总共剩下 35MB 内存可用，可问题是明摆着的，现在连续的内存块只有原来进程 B 的 20MB 和最下面可用内存 15MB。哪一块都不够进程 D 用的，这怎么办呢？



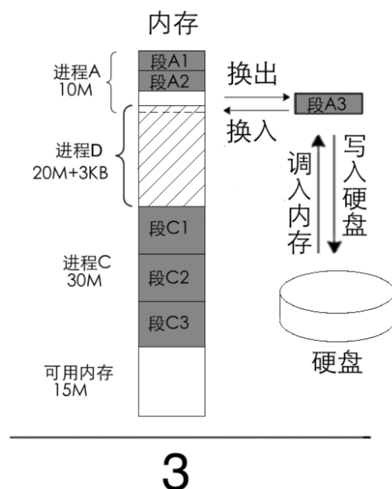
▲图 5-4　进程在分段机制下运行

两个解决方案。

- 等待进程 C 运行完后腾出内存，这样连续可用的内存就够运行进程 D 了。
- 将进程 A 的段 A3 或进程 C 的段 C1 换出到硬盘上，腾出一部分空间，加上邻接的 20MB，足够容纳进程 D。

第一个方案比较简单直接，但就是要等待，而且咱们也不知道程序 C 啥时候执行完，等个没完没了，用户还以为死机了呢，说不定一气之下就给重启了，算啦，这个方法不好，看第二个吧。

第二个方案看上去先进很多，原理是将老进程不常用的段换出到硬盘，腾出空间给新进程用，等老进程再次需要该段时，再从硬盘上将该段载入内存，如图 5-5 所示。



▲图 5-5　内存段换入换出

看上去方案完美无懈可击，虽然要用到低速的硬盘，但至少能干活。这就是当系统物理内存不足的情况下，硬盘灯会不停闪烁的原因。不过这一切需要硬件的配合才能实现，咱们一会儿介绍下这种内存管理，不过在这之前先扯点别的。

我曾经一度搞不清楚操作系统和硬件的内在联系，例如，某种功能是操作系统自己实现的，还是硬件直接支持的？甚至在更早些时候，由于知识掌握的不足，有些问题迷惑到不知该如何表达，后来才搞清楚，操作系统和硬件之间是相互依赖、相互推动、相互促进而发展起来的。比如，起初的操作系统无法对内存段做访问限制，有了这样的需求后，CPU 厂商决定采用段描述符来实现相关功能，在硬件一级上添加了 GDTR 和 LDTR 寄存器来支持全局描述符表和局部描述符表，并由硬件负责周边的安全检测。当初 CPU 硬件厂商可不是凭空造出这样一个概念的，是与操作系统厂商共同协商后才有了一套硬件方面的支持。这不仅仅在计算机行业中是这样，其他行业也一样，比如机械制造行业，如果要生产一个精度较高的零件，而目前的车床无法加工，生产车床的厂商就要提高自身水平，制造出加工精度更高的车床，而不是让零件去适应车床而降低精度。另外一个最典型的例子就是人类的直立行走，最早的时候是用四肢行走，人在思想上想把双手腾出来做其他事，所以身体便给予了“硬件”支持，慢慢发展成了只用下肢行走，这是典型的软件督促硬件发展。

虽然操作系统和 CPU 相互促进，但说到底，操作系统是软件，软件中的指令靠 CPU 来执行，如果计算机是有生命的，软件相当于思想、灵魂，而硬件才是真正的身体，思想指导身体的行为。

但并不是思想指导了所有的行为，就拿人类的运动来说，咱们的大脑产生了跑的意识后，左腿右腿就交替向前迈进。但跑起来之后，心脏会加速跳动，肺也加速了呼吸的频率，这并不是咱们主动控制的，这些器官的行为由身体里的植物神经控制。也就是说，咱们在跑步时，虽然大脑思想上只负责跑步的动作，不用向身体发命令：心脏加速、呼吸加速等，但这些器官的行为确实存在着，而且是在生理一级上自动完成不受意识控制的。

说这些就是想告诉大家，我们所写的代码仅仅是完成了某件事的一部分而已，也许是大部分，还有一部分是 CPU 硬件上负责的，这部分咱们不用管，由 CPU 自动完成。比如，调用一个函数时，CPU 自动将返回地址压入栈，进入中断时，CPU 除了压入返回地址、标志寄存器外，还要根据当前特权级决定是否压入当前栈段寄存器及指针……这样的例子太多了，不再一一列举。

东扯西扯地说了这么多后，开始说下例子中内存管理的原理，内存段是怎样被换出的。

在保护模式下，段描述符是内存段的身份证。CPU 在引用一个段时，都要先查看段描述符。很多时候，段描述符存在于描述符表中（GDT 或 LDT），但与此对应的段并不在内存中，也就是说，CPU 允许在描述符表中已注册的段不在内存中存在，这就是它提供给软件使用的策略，我们利用它实现段式内存管理。如果该描述符中的 P 位为 1，表示该段在内存中存在。访问过该段后，CPU 将段描述符中的 A 位置 1，表示近来刚访问过该段。相反，如果 P 位为 0，说明内存中并不存在该段，这时候 CPU 将会抛出个 NP（段不存在）异常，转而去执行中断描述符表中 NP 异常对应的中断处理程序，此中断处理程序是操作系统负责提供的，该程序的工作是将相应的段从外存（比如硬盘）中载入到内存，并将段描述符的 P 位置 1，中断处理函数结束后返回，CPU 重复执行这个检查，继续查看该段描述符的 P 位，此时已经为 1 了，在检查通过后，将段描述符的 A 位置 1。

以上是 CPU 加载内存段的过程，内存段是何时移出到外存上的呢？

段描述符的 A 位由 CPU 置 1，但清 0 工作可是由操作系统来完成的。此位干吗用的呢？如果仅仅用来表示该段被访问过，这也意义不大啊。其实这正是软件和硬件相互配合的体现，操作系统每发现该位为 1 后就将该位

清 0，这样一来，在一个周期内统计该位为 1 的次数就知道该段的使用频率了，从而可以找出使用频率最低的段。当物理内存不足时，可以将使用频率最低的段换出到硬盘，以腾出内存空间给新的进程。当段被换出到硬盘后，操作系统将该段描述符的 P 位置 0。当下次这个进程上 CPU 运行后，如果访问了这个段，这样程序流就回到了刚开始 CPU 检查出 P 位为 0、紧接着抛出异常、执行操作系统中断处理程序、换入内存段的循环。

另外，内存中的数据是二进制的，段被换出到硬盘上也以二进制形式存储，数据内容都是一样的，只是存储介质不同而已，不要因为陌生而觉得段的换入换出深不可测，这无非是一段二进制数据在内存和外存之间拷贝来拷贝去而已，其过程就像将一个 txt 文件读到内存中修改后再保存到硬盘一样。

第二个方法虽然解决了内存不足的问题，但也有缺陷。比如物理内存特别小，无法容纳任何一个进程的段，这就没法运行进程了，更没法做段的换入换出。也许有人会说，这是用户的问题，这么小的内存还拿出来用，这不是“逗比”吗？您还别说，一会儿介绍的内存分页机制，理论上只要 4KB 内存就可以让程序运行下去。另外一种情况是若进程的段比较大，换出时要将整个段全部搬到外存上，这种 IO 操作太多了机器响应奇慢无比，用户是无法接受的。还有没有更好的方法呢？

想一想，出现这种问题的原因是什么？问题的本质是在目前只分段的情况下，CPU 认为线性地址等于物理地址。而线性地址是由编译器编译出来的，它本身是连续的，所以物理地址也必须要连续才行，但我们可用的物理地址不连续。换句话说，如果线性地址连续，而物理地址可以不连续，不就解决了吗。

按照这种思路，我们首先要做的是解除线性地址与物理地址一一对应的关系，然后将它们的关系重新建立。通过某种映射关系，可以将线性地址映射到任意物理地址。

有很多实现映射的方法，比如可以写个哈希算法，将线性地址做 key，而 value 是物理地址。不过，这都是软件实现的算法，时间复杂度再低，效率肯定不如硬件“短、平、快”，因为硬件中的操作更直接，并且已经在电路上做过优化，而软件的效率主要取决于代码的算法和编译器的优化能力，即使能产生出最优的机器码，也是被当作普通指令处理：先要到内存中取指，译码，再执行，不说别的，就光是取指这步就已经很慢了，毕竟内存在 CPU 眼里是低速设备。所以，对于地址转换这种实时性较高的需求，CPU 已经给予了我们最大的硬件支持，在 CPU 实现中，这种映射关系是通过一张表来实现的，该表就是我们所说的页表，查找页表的工作也是由硬件完成的。这张表是什么样的呢？我们在下一节中给出答案。

5.2.2　一级页表

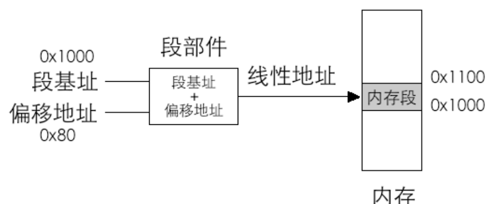
为了给大家说清楚分页机制，我们先从宏观上说一下 CPU 地址变换过程，先让大家有个直观的印象。

分页机制其实是建立在分段机制之上的。这是不是有些让大家意外呢？其实这并不是说分页机制依赖于分段机制，只是这内存分段机制属于 Intel IA32 架构骨子里的东西，改是改不掉的，除非从头再造个 CPU 出来，所以分页机制只能在现有分段机制大局已定的情况下诞生，它们两者的关系是怎样的呢？让我们先从保护模式下的分段机制开始梳理。

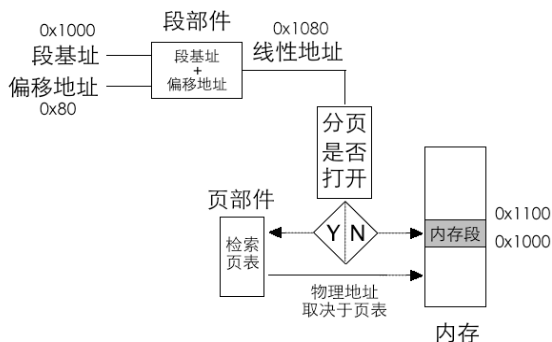
尽管在保护模式中段寄存器中的内容已经是选择子，但选择子最终就是为了要找到段基址，其内存访问的核心机制依然是“段基址：段内偏移地址”，这两个地址在相加之后才是绝对地址，也就是我们所说的线性地址，此线性地址在分段机制下被 CPU 认为是物理地址，直接拿来就能用，也就是说，此线性地址可以直接送上地址总线。将段基址和段内偏移地址相加求和的工作是由 CPU 的段部件自动完成的。整个访问内存的过程如图 5-6 所示。

分页机制要建立在图 5-6 所示分段机制的基础上，也就是说，段部件的工作依然免不了，所以，分页只能是在分段之后进行的，其过程如图 5-7 所示。

图 5-7 说明，CPU 在不打开分页机制的情况下，是按照默认的分段方式进行的，段基址和段内偏移地址经过段部件处理后所输出的线性地址，CPU 就认为是物理地址。如果打开了分页机制，段部件输出的线性地址就不再等同于物理地址了，我们称之为虚拟地址，它是逻辑上的，是假的，不应该被送上地址总线（因为地址只是个数字，任何数字都可以当作地址，这里说的“不应该”是指应该人为保证送上地址总线上的数字是正确的地址）。CPU 必须要拿到物理地址才行，此虚拟地址对应的物理地址需要在页表中查找，这项查找工作是由页部件自动完成的。为了要搞清楚页部件的工作原理，必须要搞清楚这两件事。



▲图 5-6　内存分段机制下地址访问



▲图 5-7　分页机制

（1）分页机制的原理。

（2）页表的结构。

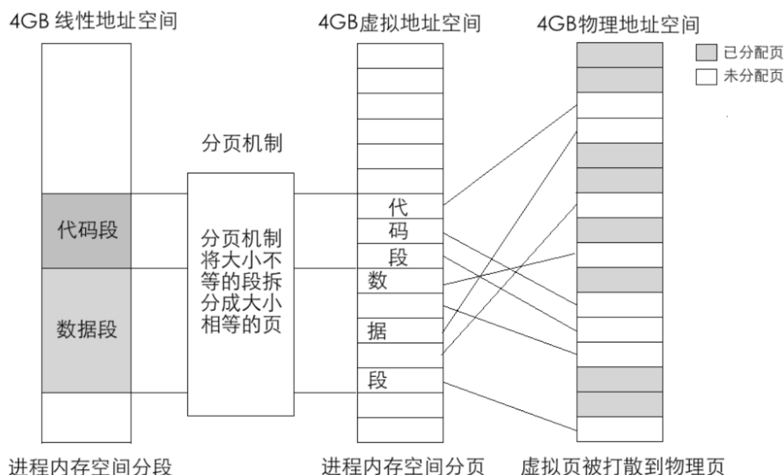
下面我们将从这两方面入手，循序渐进地展开分页机制原理。

经过段部件处理后，保护模式的寻址空间是 4GB，注意啦，这个寻址空间是指线性地址空间，它在逻辑上是连续的。分页机制的思想是：通过映射，可以使连续的线性地址与任意物理内存地址相关联，逻辑上连续的线性地址其对应的物理地址可以不连续。

分页机制的作用有两方面。

- 将线性地址转换成物理地址。
- 用大小相等的页代替大小不等的段。

这两方面的作用如图 5-8 所示。



▲图 5-8　分页机制的作用

由于有了线性地址到真实物理地址的这层映射，经过段部件输出的线性地址便有了另外一个名字，虚拟地址。

下面根据图 5-8 给大家介绍下操作系统在分页机制下加载进程的过程。

图 5-8 表示的是一个进程的地址转换过程，从线性空间到虚拟空间再到物理地址空间，每个空间大小都是 4GB。图上的 4GB 物理地址空间属于所有进程包括操作系统在内的共享资源，其中标注为已分配页的内存块被分配给了其他进程，当前进程只能使用未分配页。此转换过程对任意一个进程都是一样的，也就是说，每个进程都有自己的 4GB 虚拟空间。

前面说过啦，分页机制建立在分段机制之上，与其脱离不了干系，即使在分页机制下的进程也要先经过逻辑上的分段才行，每加载一个进程，操作系统按照进程中各段的起始范围，在进程自己的 4GB 虚拟地址空间中寻

找可用空间分配内存段，此虚拟地址空间可以是页表，也可以是操作系统维护的某种数据结构，总之此阶段的分配是逻辑上的，并没有真正写入物理内存。在分页机制下，分配情形如图 5-8 中所示的虚拟地址空间中的代码段和数据段。代码段和数据段在逻辑上被拆分成以页为单位的小内存块。这时的虚拟地址虚如其名，不能存放任何数据。接着操作系统开始为这些虚拟内存页分配真实的物理内存页，它查找物理内存中可用的页，然后在页表中登记这些物理页地址，这样就完成了虚拟页到物理页的映射，每个进程都以为自己独享 4GB 地址空间。

以上在宏观上笼统地介绍了分页机制下操作系统加载用户进程的整个流程，先让大家心中有数，了解我们下面所说的内容是什么。也许您对此过程并不十分理解，不过没关系，下面咱们开始从头说起。

映射这个概念大家应该比较清楚，对应的英文单词是 map，意为地图。地图是对实际地理空间的一种抽象，地图上的每个位置都代表某个真实地理空间，这种地图上与地理上一一对应的关系就称为映射。

在内存地址中，最简单的映射方法是逐字节映射，即一个线性地址对应一个物理地址。比如线性地址为 0x0，其对应的物理地址可以是 0x0、0x10 或其他你喜欢的数字，若线性地址为 0x1，对应的物理地址为 0x1、0x11 或其他你喜欢的数字。我们需要找个地方来存储这种映射关系，这个地方就是页表（Page Table）。页表就是个 N 行 1 列的表格，页表中的每一行（只有一个单元格）称为页表项（Page Table Entry，PTE），其大小是 4 字节，页表项的作用是存储内存物理地址。当访问一个线性地址时，实际上就是在访问页表项中所记录的物理内存地址。

页表与物理内存关系示意如图 5-9 所示，为了不误导大家，提前声明，图 5-9 仅仅是线性地址与物理地址逐字节映射的示意图，真正的页表不是逐字节映射，图 5-9 仅为展示页表与物理地址映射原理，我们要循序渐进嘛。

如果采用这种线性地址与物理地址一一映射的方案。

（1）表中就应该有 4G 个页表项。

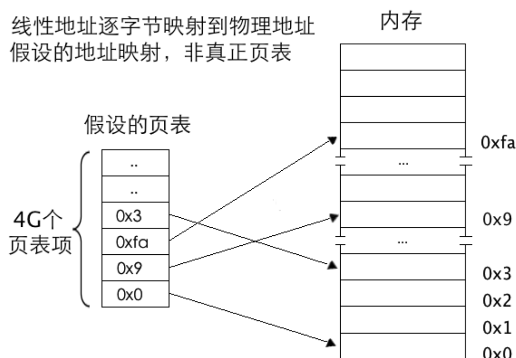
（2）32 位的地址要用 4 字节的页表项来存储，页表总共大小是 4Byte*4G =16GB。

分页机制本质上是将大小不同的大内存段拆分成大小相等的小内存块。以上方案其实就是将 4GB 空间划分成 4G 个内存块，每个内存块大小是 1 字节。页表也是存储在内存中的，为了表示 32 位地址，每个页表项必须要 4 字节，若按此方案，光是页表就要占 16GB 内存，得不偿失，显然方案不合理。

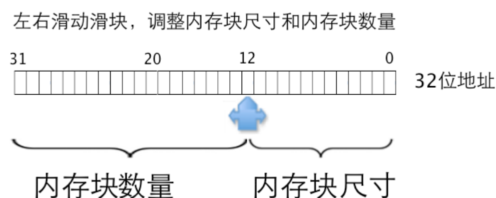
以上方案不成立的原因是内存块数量太大了，也就是说，在总的 4GB 地址空间恒定不变的情况下，内存块尺寸选的太小了。为了找到合适的内存块大小，我们做下列分析与尝试。

任意进制的数字都可以分成高位部分和低位部分，若将低位部分理解为单位大小，高位部分则是这种单位的数量。例如六万的十进制可表示为 60000，也可以表示为 60 千，也就是将 60000 分成高位 60 和低位 1000 两部分。

32 位地址表示 4GB 空间，可以将 32 位地址分成高低两部分，低地址部分是内存块大小，高地址部分是内存块数量，它们是这样一种关系：内存块数*内存块大小=4GB。故我们可以在图 5-10 所示的 32 位地址上滑动滑块找到合适的内存块尺寸。滑块右边是内存块尺寸，滑块左边是内存块数量。



▲图 5-9　页表与物理内存关系示意

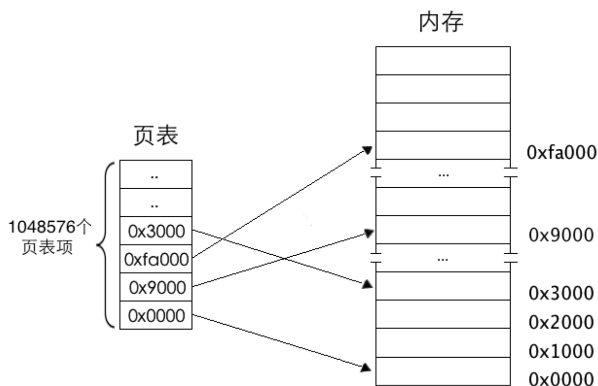


▲图 5-10　寻找合适的页尺寸

为了节省页表空间，势必要将滑块往左调整，以使内存块尺寸变大，这样内存块数量变小，从而减少了页表

项数量。如果滑块指向第 20 位，内存块大小为 2 的 20 次方，即 1MB，内存块数量为 2 的 12 次方，即 4K 个。若滑块指向第 12 位，内存块大小则为 2 的 12 次方，即 4KB，内存块数量则为 2 的 20 次方，1M，即 1048576 个。这里所说的内存块，其官方名称是页，CPU 中采用的页大小恰恰就是 4KB，也就是图 5-10 中滑块的落点处。

页是地址空间的计量单位，并不是专属物理地址或线性地址，只要是 4KB 的地址空间都可以称为一页，所以线性地址的一页也要对应物理地址的一页。一页大小为 4KB，这样一来，4GB 地址空间被划分成 4GB/4KB=1M 个页，也就是 4GB 空间中可以容纳 1048576 个页，页表中自然也要有 1048576 个页表项，这就是我们要说的一级页表。一级页表如图 5-11 所示。



▲图 5-11　页表及页表项

图 5-11 所示是一级页表模型，由于页大小是 4KB，所以页表项中的物理地址都是 4k 的整数倍，故用十六进制表示的地址，低 3 位都是 0。就拿第 3 个页表项来说，其值为 0x3000，表示该页对应的物理地址是 0x3000。

可能，您心里一直有个疑问：页表如何使用呢？也就是如何将线性地址转换成物理地址呢？

还是用图 5-10 帮助理解，滑块正落到在 32 位地址的第 12 位。右边第 11～0 位用来表示页的大小，也就是这 12 位可以作为页内寻址。左边第 31～12 位用来表示页的数量，同样这 20 位也用来索引一个页（索引范围 0～0xfffff），表示第几个页，对吧。

其实也可以这样理解：任意一个地址最终会落到某一个物理页中。32 位地址空间共有 1M（1048756）个物理页，首先要做的是定位到某个具体物理页，然后给出物理页内的偏移量就可以访问到任意 1 字节的内存啦。所以，用 20 位二进制就可以表示全部物理页啦。标准页都是 4KB，12 位二进制便可以表达 4KB 之内的任意地址。

在 32 位保护模式下任何地址都是用 32 位二进制表示的，包括虚拟地址也是。经以上分析，虚拟地址的高 20 位可用来定位一个物理页，低 12 位可用来在该物理页内寻址。这是如何实现的呢？物理地址写在页表的页表项中，段部件输出的只是线性地址，所以问题就变成了：怎样用线性地址找到页表中对应的页表项。

在此之前，大家要知道两件事。

（1）分页机制打开前要将页表地址加载到控制寄存器 cr3 中，这是启用分页机制的先决条件之一，在介绍二级页表时会细说。所以，在打开分页机制前加载到寄存器 cr3 中的是页表的物理地址，页表中页表项的地址自然也是物理地址了。

（2）虽然内存分页机制的作用是将虚拟地址转换成物理地址，但其转换过程相当于在关闭分页机制下进行，过程中所涉及到的页表及页表项的寻址，它们的地址都被 CPU 当作最终的物理地址（本来也是物理地址）直接送上地址总线，不会被分页机制再次转换（否则会递归转换下去）。

刚才说过啦，如何通过线性地址找到其对应的页表项才是转换的关键。既然页表位于内存中，所以只要提供页表项的物理地址便能够访问到页表项。页表本身属于线性表结构，相当于页表项数组，访问其中任意页表项成员，只要知道该表页项的索引（下标）就够了。

分析过后，地址转换过程原理如下。

一个页表项对应一个页，所以，用线性地址的高 20 位作为页表项的索引，每个页表项要占用 4 字节大小，所以这高 20 位的索引乘以 4 后才是该页表项相对于页表物理地址的字节偏移量。用 cr3 寄存器中的页表物理地址加上此偏移量便是该页表项的物理地址，从该页表项中得到映射的物理页地址，然后用线性地址的低 12 位与该物理页地址相加，所得的地址之和便是最终要访问的物理地址。

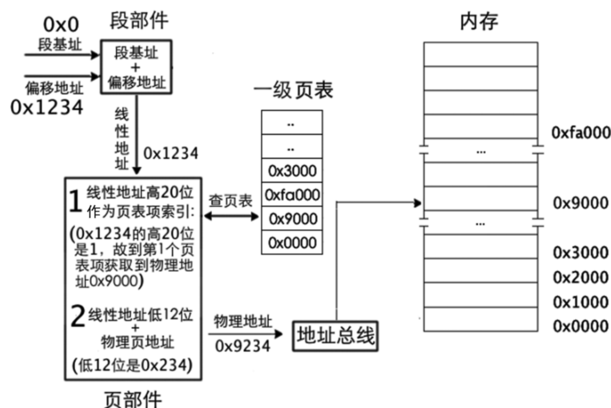
曾经有同学对地址转换过程感到迷惑，误以为启用分页后，页表项地址也是虚拟地址，还需要被转换，

转换过程无限递归下去，这显然是不对的。

以上所说的转换步骤多少都有点麻烦，既然地址转换算法已经是固定的了，何不使其在硬件一级自动完成呢？有道理，所以 CPU 中集成了专门用来干这项工作的硬件模块，我们把该模块称为页部件。当程序中给出一个线性地址时，页部件分析线性地址，按照以上算法，自动在页表中检索到物理地址。

总结一下页部件的工作：用线性地址的高 20 位在页表中索引页表项，用线性地址的低 12 位与页表项中的物理地址相加，所求的和便是最终线性地址对应的物理地址。

咱们还是通过例子来理解转换过程吧。拿 mov ax，[0x1234]来说，其地址转换完整过程如图 5-12 所示。



▲图 5-12　通过一级页表将线性地址转换成物理地址过程

假设咱们是在平坦模型下工作，不管段选择子值是多少，其所指向的段基址都是 0，指令 mov ax，[0x1234]中的 0x1234 称为有效地址，它作为“段基址：段内偏移地址”中的段内偏移地址。这样段基址为 0，段内偏移地址为 0x1234，经过段部件处理后，输出的线性地址是 0x1234。由于咱们是演示分页机制，必须假定系统已经打开了分页机制，所以线性地址 0x1234 被送入了页部件。页部件分析 0x1234 的高 20 位，用十六进制表示高 20 位是 0x00001。将此项作为页表项索引，再将该索引乘以 4 后加上 cr3 寄存器中页表的物理地址，这样便得到索引所指代的页表项的物理地址，从该物理地址处（页表项中）读取所映射的物理页地址：0x9000。线性地址的低 12 位是 0x234，它作为物理页的页内偏移地址与物理页地址 0x9000 相加，和为 0x9234，这就是线性地址 0x1234 最终转换成的物理地址。

一级页表说了这么多，完全是为了讲述页表原理，这样就能更好地理解下面要讲的二级页表，它们在原理上一脉相承。因为目前现代操作系统一般都是用的二级页表，咱们的系统也采用二级页表，下一节咱们再见啦。

5.2.3　二级页表

前面讲述了页表的原理，并以一级页表作为原型讲述了地址转换过程。既然有了一级页表，为什么还要搞个二级页表呢？理由如下。

（1）一级页表中最多可容纳 1M（1048576）个页表项，每个页表项是 4 字节，如果页表项全满的话，便是 4MB 大小。

（2）一级页表中所有页表项必须要提前建好，原因是操作系统要占用 4GB 虚拟地址空间的高 1GB，用户进程要占用低 3GB。

（3）每个进程都有自己的页表，进程一多，光是页表占用的空间就很可观了。

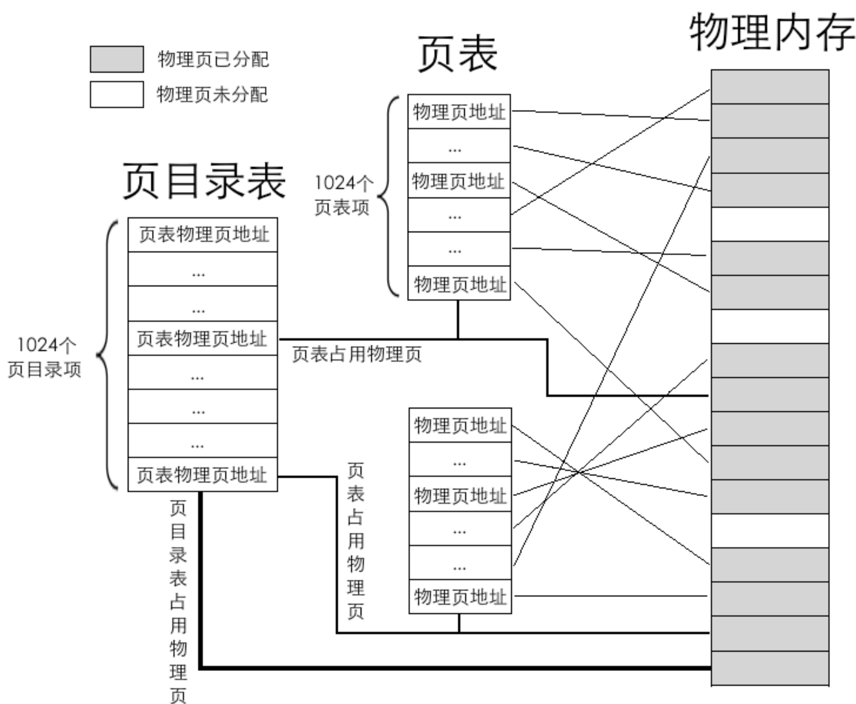
归根结底，我们要解决的是：不要一次性地将全部页表项建好，需要时动态创建页表项。如何解决呢？

二级页表很好地解决了该问题。我们来说下什么是二级页表。

无论是几级页表，标准页的尺寸都是 4KB，这一点是不变的。所以 4GB 线性地址空间最多有 1M 个标准页。一级页表是将这 1M 个标准页放置到一张页表中，二级页表是将这 1M 个标准页平均放置 1K 个页表中。每个页表中包含有 1K 个页表项。页表项是 4 字节大小，页表包含 1K 个页表项，故页表大小为 4KB，这恰恰是一个标准页的大小。

拆分出了这么多个页表，如何使用它们呢？为此，专门有个页目录表来存储这些页表。每个页表的物理地址在页目录表中都以页目录项（Page Directory Entry，　PDE）的形式存储，页目录项大小同页表项一样，都用来描述一个物理页的物理地址，其大小都是 4 字节，而且最多有 1024 个页表，所以页目录表也是 4KB 大小，同样也是标准页的大小。

页表是用于管理内存的数据结构，其也要占用内存，所以页目录表和页表所占用的物理页，同样混迹于物理内存之中，如图 5-13 所示。



▲图 5-13　二级页表占用内存示意

图 5-13 中，页目录表中共 1024 个页表，也就是有 1024 个页目录项。一个页目录项中记录一个页表物理页地址，物理页地址是指页的物理地址，在页目录项及页表项中记录的都是页的物理地址，页大小都是 0x1000，即 4096，因此页地址是以 000 为结尾的十六进制数字。每个页表中有 1024 个页表项，每个页表项中是一个物理页地址，最终数据写在这页表项中指定的物理页中。页表项中分配的物理页地址在真正物理内存中离散分布，毫无规律可言，操作系统负责这些物理页的分配与释放。由于页目录表和页表本身都要占用内存，且为 4KB 大小，故它们也会由操作系统在物理内存中分配一物理页存放。图中最粗的线存放页目录表物理页，稍细一点的线指向的是用来存放页表的物理页，其他最细的线是页表项中分配的物理页，页表结构本身与其他数据混布渗透在物理内存中，页表所占用的物理页在外在形式上与其他数据占用的物理页没有什么不同，只有 CPU 知道它们的作用不同。页表在建立之初，物理内存各部分的布局还是相对较整洁的，随着操作系统分配或释放内存的动作越来越频繁，物理内存的布局将更加零散。

二级页表与一级页表在原理上相同，但结构上已经有了很大不同，它们在虚拟地址到物理地址转换方法上也有很大不同。

我们已经知道，前面所说的一级页表转换方法，是将 32 位虚拟地址拆分成两部分，高 20 位用于定位一个物

理页，低 12 位用于物理页内的偏移量。在二级页表转换中，依然用 32 位虚拟地址的不同部分来定位物理页。

在二级页表是这样的：

每个页表中可容纳 1024 个物理页，故每个页表可表示的内存容量是 1024*4KB=4MB。页目录中共有 1024 个页表，故所有页表可表示的内存容量是 1024*4MB=4GB，这已经达到了 32 位地址空间的最大容量。所以说，任意一个 32 位物理地址，它必然在某个页表之内的某个物理页中。我们定位某一个物理页，必然要先找到其所属的页表。页目录中 1024 个页表，只需要 10 位二进制就能够表示了，所以，虚拟地址的高 10 位（第 31～22 位）用来在页目录中定位一个页表，也就是这高 10 位用于定位页目录中的页目录项 PDE，PDE 中有页表物理页地址。找到页表后，到底是页表中哪一个物理页呢？由于页表中可容纳 1024 个物理页，故只需要 10 位二进制就能够表示了。所以虚拟地址的中间 10 位（第 21～12 位）用来在页表中定位具体的物理页，也就是在页表中定位一个页表项 PTE，PTE 中有分配的物理页地址。由于标准页都是 4KB，12 位二进制便可以表达 4KB 之内的任意地址，故线性地址中余下的 12 位（第 11～0 位）用于页内偏移量。

经以上分析，二级页表地址转换原理是将 32 位虚拟地址拆分成高 10 位、中间 10 位、低 12 位三部分，它们的作用是：高 10 位作为页表的索引，用于在页目录表中定位一个页目录项 PDE，页目录项中有页表物理地址，也就是定位到了某个页表。中间 10 位作为物理页的索引，用于在页表内定位到某个页表项 PTE，页表项中有分配的物理页地址，也就是定位到了某个物理页。低 12 位作为页内偏移量用于在已经定位到的物理页内寻址。

同一级页表一样，访问任何页表内的数据都要通过物理地址。由于页目录项 PDE 和页表项 PTE 都是 4 字节大小，给出了 PDE 和 PTE 索引后，还需要在背后悄悄乘以 4，再加上页表物理地址，这才是最终要访问的绝对物理地址。转换过程背后的具体步骤如下。

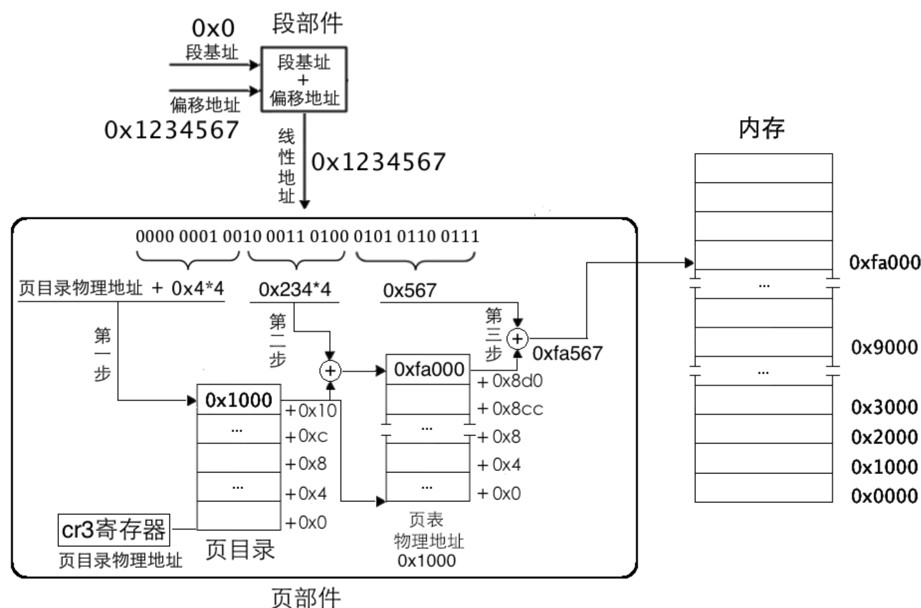
（1）用虚拟地址的高 10 位乘以 4，作为页目录表内的偏移地址，加上页目录表的物理地址，所得的和，便是页目录项的物理地址。读取该页目录项，从中获取到页表的物理地址。

（2）用虚拟地址的中间 10 位乘以 4，作为页表内的偏移地址，加上在第 1 步中得到的页表物理地址，所得的和，便是页表项的物理地址。读取该页表项，从中获取到分配的物理页地址。

（3）虚拟地址的高 10 位和中间 10 位分别是 PDE 和 PTE 的索引值，所以它们需要乘以 4。但低 12 位就不是索引值啦，其表示的范围是 0～0xfff，作为页内偏移最合适，所以虚拟地址的低 12 位加上第 2 步中得到的物理页地址，所得的和便是最终转换的物理地址。

这种自动化较强的工作，还是由页部件自动完成的。

还是举例子来说，比如 mov ax，[0x1234567]。其过程如图 5-14 所示。



▲图 5-14　二级页表虚拟地址到物理地址转换

图 5-14 中页目录项、页表项中的地址值都是为演示而虚构的，咱们对照着图示细说下转换过程。

平坦模型下段基址为 0，指令 mov ax，[0x1234567]，经过段部件处理，输出的线性地址为 0x1234567，由于是在分页机制下，此地址被认为是虚拟地址，需要被页部件转换。页部件首先要把地址拆分成高 10 位、中间 10 位、低 12 位三部分。其实低 12 位最容易得出，十六进制的每 1 位代表 4 位二进制，所以低 12 位直接就是 0x567。高 10 位和中间 10 位，不容易一眼看出来，所以还是将它们换算成二进制看比较容易。

0x1234567 的二进制形式是：0000 0001 0010 0011 0100 0101 0110 0111。

高 10 位是 0000 0001 00，十六进制为 0x4。

中间 10 位是 10 0011 0100，十六进制为 0x234。

低 12 位是 0101 0110 0111，十六进制为 0x567。

第一步：为了得到页表物理地址，页部件用虚拟地址高 10 位乘以 4 的积与页目录表物理地址相加，所得的和便是页目录项地址，读取该页目录项，获取页表物理地址。这里是 0x4*4=0x10，页表物理地址存储在 cr3 寄存器中，由于是过程演示，无需给出具体数值，我们只需要用 0x10 作为页表中的偏移地址便能够找到对应的页目录项。如图页目录表和页表中，对于内存地址我们给出的都是偏移量，所以地址前都有个加号“+”。这里我们找到了最上面的页目录项，其值为 0x1000。这意味着要找的页表位于物理地址 0x1000。

第二步：为了得到具体的物理页，需要找到页表中对应的页表项。页部件用虚拟地址中间 10 位的值乘以 4 的积与第一步中得到的页表地址相加，所得的和便是页表项物理地址。这里是 0x234*4=0x8d0，页表项物理地址是 0x8d0+0x1000=0x18d0。在该页表项中的值是 0xfa000，这意味着分配的物理页地址是 0xfa000。

第三步：为了得到最终的物理地址，用虚拟地址低 12 位作为页内偏移地址与第二步中得到的物理页地址相加，所得的和便是最终的物理地址。这里是 0xfa000+0x567=0xfa567。

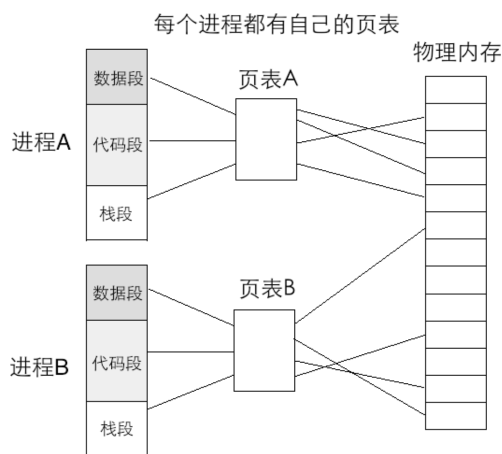
经过这三步后，页部件将虚拟地址 0x1234567 转换成物理地址 0xfa567。

图 5-14 中，页目录的页目录项，页表中的页表项，它们中记录的物理地址都是随意写上去的，纯粹是为演示，所以大家不要纠结它们是怎么来的。咱们以后在建立页表时，会在页目录项及页表项中写入合适的值，所谓“合适”，是指提前设计好的，马上咱们就知道了。

每个任务都有自己的页表，内存是程序竞技的江湖，这样一来，每个任务都以为自己“独霸武林”，活在自己的虚拟地址空间中。图 5-15 所示是多个进程并行时，在各自虚拟空间中的情况。

另外，任务在切换时，页表也需要跟着切换，我们将在加载进程时讲述有关部分。

到了现在，我们该说说页目录项和页表项的事啦，一直都说它们是 4 字节大小，用来存储物理页地址，但一直未曾见过它们的真实面目。下面给大家呈上它们的结构，如图 5-16 所示。



▲图 5-15　每个进程都有自己的虚拟空间



▲图 5-16　页目录项及页表项

您看，它们确实如之前所述，4 字节大小，但其内容并不全是物理地址，只有第 12～31 位才是物理地址，这才 20 位。按理说 32 位地址应该用 32 位来表示啊，否则不就误差严重了吗。是这样的，因为页

目录项和页表项中的都是物理页地址，标准页大小是 4KB，故地址都是 4K 的倍数，也就是地址的低 12 位是 0，所以只需要记录物理地址高 20 位就可以啦。这样省出来的 12 位（第 0～11 位）可以用来添加其他属性，下面对这些属性从低到高逐位介绍。

P，Present，意为存在位。若为 1 表示该页存在于物理内存中，若为 0 表示该表不在物理内存中。操作系统的页式虚拟内存管理便是通过 P 位和相应的 pagefault 异常来实现的。

RW，Read/Write，意为读写位。若为 1 表示可读可写，若为 0 表示可读不可写。

US，User/Supervisor，意为普通用户/超级用户位。若为 1 时，表示处于 User 级，任意级别（0、1、2、3）特权的程序都可以访问该页。若为 0，表示处于 Supervisor 级，特权级别为 3 的程序不允许访问该页，该页只允许特权级别为 0、1、2 的程序可以访问。

PWT，Page-level Write-Through，意为页级通写位，也称页级写透位。若为 1 表示此项采用通写方式，表示该页不仅是普通内存，还是高速缓存。此项和高速缓存有关，“通写”是高速缓存的一种工作方式，本位用来间接决定是否用此方式改善该页的访问效率。这里咱们直接置为 0 就可以啦。

PCD，Page-level Cache Disable，意为页级高速缓存禁止位。若为 1 表示该页启用高速缓存，为 0 表示禁止将该页缓存。这里咱们将其置为 0。

A，Accessed，意为访问位。若为 1 表示该页被 CPU 访问过啦，所以该位是由 CPU 设置的。还记得段描述符中的 A 位和 P 位吗？这两位在一起可以实现段式虚拟内存管理。和它们一样，这里页目录项和页表项中的 A 位也可以用来记录某一内存页的使用频率（操作系统定期将该位清 0，统计一段时间内变成 1 的次数），从而当内存不足时，可以将使用频率较低的页面换出到外存（如硬盘），同时将页目录项或页表项的 P 位置 0，下次访问该页引起 pagefault 异常时，中断处理程序将硬盘上的页再次换入，同时将 P 位置 1。

D，Dirty，意为脏页位。当 CPU 对一个页面执行写操作时，就会设置对应页表项的 D 位为 1。此项仅针对页表项有效，并不会修改页目录项中的 D 位。

PAT，Page Attribute Table，意为页属性表位，能够在页面一级的粒度上设置内存属性。比较复杂，将此位置 0 即可。

G,Global，意为全局位。由于内存地址转换也是颇费周折，先得拆分虚拟地址，然后又要查页目录，又要查页表的，所以为了提高获取物理地址的速度，将虚拟地址与物理地址转换结果存储在 TLB（Translation Lookaside Buffer）中，TLB 以后咱们会细说。在此先知道 TLB 是用来缓存地址转换结果的高速缓存就 ok 啦。此 G 位用来指定该页是否为全局页，为 1 表示是全局页，为 0 表示不是全局页。若为全局页，该页将在高速缓存 TLB 中一直保存，给出虚拟地址直接就出物理地址啦，无需那三步骤转换。由于 TLB 容量比较小（一般速度较快的存储设备容量都比较小），所以这里面就存放使用频率较高的页面。顺便说一句，清空 TLB 有两种方式，一是用 invlpg 指令针对单独虚拟地址条目清理，或者是重新加载 cr3 寄存器，这将直接清空 TLB。

AVL，意为 Available 位，表示可用，谁可以用？当然是软件，操作系统可用该位，CPU 不理会该位的值，那咱们也不理会吧。

似乎关于页表的部分已经说了不少了，也许大伙儿都已经等不及了，在想究竟什么时候才能启用页表。前面我们讲述的原理比较多，占的篇幅比较长，俗话说，磨刀不误砍柴工嘛。在我们刚刚讲述完了页目录项及页表项后，实际上我们只完成了启用分页机制的第一步，我们距启用分页机制还有一些工作要做，后面的都是小事，用不了多少篇幅。

启用分页机制，我们要按顺序做好三件事。

（1）准备好页目录表及页表。

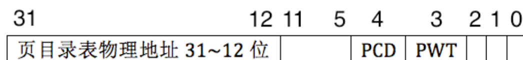
（2）将页表地址写入控制寄存器 cr3。

（3）寄存器 cr0 的 PG 位置 1。

在介绍完了页目录项及页表项后，我们便知道如何创建一个页目录表和页表了，这相当于我们已经有能力完成第一步。下面咱们看看后面的两部分。

页表同描述符表一样，是个内存中的数据结构，处理器要使用它们，必须要知道它们的物理地址，所以页表也有个专门的寄存器来存储其地址。其实这就是前面咱们多次提到的众多控制寄存器（目前处理器

中的控制寄存器有 cr0～cr7）中的一个：cr3 寄存器。控制寄存器 cr3 用于存储页表物理地址，所以 cr3 寄存器又称为页目录基址寄存器（Page Directory Base Register，PDBR）。其结构如图 5-17 所示。



▲图 5-17　页目录基址寄存器 PDBR（控制寄存器 cr3）

由于页目录表所在的地址要求在一个自然页内，即页目录的起始地址是 4KB 的整数倍，低 12 位地址全是 0。所以，只要在 cr3 寄存器的第 31～12 位中写入物理地址的高 20 位就行了。另外，cr3 寄存器的低 12 位中，除第 3 位的 PWT 位和第 4 位的 PCD 位外，其余位都没用。PWT 位和 PCD 位在介绍页表项时说过了，它们用于设置高速缓存相关的特性，在此将其置为 0 即可。这样一来低 12 位全部为 0，故只需要把页目录表物理地址的高 20 位写入 cr3 寄存器即可。

因为控制寄存器是可以与通用寄存器互相传递数据的，所以为 cr3 寄存器赋值则没有那么复杂，可以用现成的 mov 命令，mov 指令中控制寄存器与通用寄存器互传数据的格式是：mov cr[0～7]，r32 或 mov r32，cr[0～7]。

坚持一下，离启用分页机制，我们只差一步啦。

前两步是打开分页机制的铺垫，现在看第 3 步。启动分页机制的开关是将控制寄存器 cr0 的 PG 位置 1，PG 位是 cr0 寄存器的最后一位：第 31 位。如果大家忘记了 cr0 寄存器结构，请参见图 4-10 控制寄存器 CR0。PG 位为 1 后便进入了内存分页运行机制，段部件输出的线性地址成为虚拟地址（顺便说一下，第 0 位是 PE 位，用来进入保护模式的开关）。在将 PG 位置 1 之前，系统都是在内存分段机制下工作，段部件输出的线性地址便直接是物理地址，也就意味着在第 2 步中，cr3 寄存器中的页表地址是真实的物理地址。

好啦，有关页表的部分，真的到此为止啦，说完了，咱们必须要动手实践一下啦。

5.2.4　规划页表之操作系统与用户进程的关系

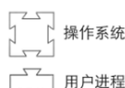
分页的第一步要准备好一个页表，我们的页表是什么样子呢？现在我们要设计一个页表啦。

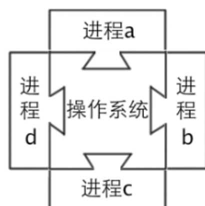
设计页表其实就是设计内存布局，不过在规划内存布局之前，我们需要了解用户进程与操作系统之间的关系。

前面讲保护模式时，我们知道，为了计算机安全，用户进程必须运行在低特权级，当用户进程需要访问硬件相关的资源时，需要向操作系统申请，由操作系统去做，之后将结果返回给用户进程。进程可以有无限多个，而操作系统只有一个，所以，操作系统必须“共享”给所有用户进程。它们的关系如图 5-18 所示。

图 5-18 不仅展示了用户进程共享操作系统的逻辑依赖关系，还用插槽展示了它们的配合关系，用户进程要想完成某件工作，需要与操作系统结合在一起才行，那用户进程和操作系统它们是什么关系呢？

要完成一件事，用户进程做的事情只能算个半成品，您可以理解成：用户的代码加上所需要的操作系统中的部分代码才算完整的程序，为什么说是操作系统中的部分代码呢？原因很简单，因为操作系统严格来说是一套功能的集合，用户进程所需要的部分可能仅仅是其中的一小部分，并不是所有功能都会用到。用户进程能用哪些功能，是由操作系统决定的，不是用户想用什么就用什么，而是操作系统提供什么它就用什么。完整的程序概念如图 5-19 所示。





1 用户进程共享操作系统
2 用户进程与操作系统配合“在一起”才能完成工作

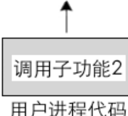
▲图 5-18　进程共享操作系统

操作系统代码



完整的程序

系统调用接口



用户进程代码

用户程序代码 + 所调用的操作系统代码 = 完整的程序

▲图 5-19　完整的程序

它和操作系统需要共同配合才能完成一件事，它们的关系有如服务提供商和客户的关系。服务提供商

提供一些服务，客户只能用这些服务，也就是说客户依赖于服务提供商提供的服务项目，是服务提供商的主导客户。比如咱们在网上买东西，咱们只需要挑选好商品后写好地址，然后下订单就成了。这事完了吗？必须没有，得拿到商品才算完事。所以，之后的事情就交给电商了，他们为你从库中挑选商品，然后用物流送到您家，这才拿到了商品，到此才算完事了。

以上购物的例子就是典型的用户程序和操作系统的关系，咱们挑商品下单这件事就相当于进程，而网上的电商才是充当了操作系统的角色，根据买家的需求找到所需要的资源，然后通过物流，将商品（结果）返回。

上述所说的用户进程和操作系统的关系，都是基于用户进程共享操作系统的。我们设计的页表也要满足这个基本要求：共享。

如何在页表中实现共享呢？这个简单，只要操作系统属于用户进程的虚拟地址空间就好了。

说起来简单，这该怎么做呢？我们可以把 4GB 虚拟地址空间分成两部分，一部分专门划给操作系统，另一部分就归用户进程使用。比如我们之前都听说过，操作系统在 4GB 内存的高地址，用户进程在 4GB 内存的低地址。比如 Linux，它就运行在虚拟地址的 3GB 以上，其他用户进程都运行在 3GB 以下。

页表的设计是要根据内存分布情况来决定的，我们也学习 Linux 的作法，在用户进程 4GB 虚拟地址空间的高 3GB 以上的部分划分给操作系统，0～3GB 是用户进程自己的虚拟空间。为了实现共享操作系统，让所有用户进程 3GB～4GB 的虚拟地址空间都指向同一个操作系统，也就是所有进程的虚拟地址 3GB～4GB 本质上都是指向的同一片物理页地址，这片物理页上是操作系统的实体代码。实现起来也比较容易，只要保证所有用户进程虚拟地址空间 3GB～4GB 对应的页表项中所记录的物理页地址是相同的就行啦。哈哈，这句话确实有点长，我自己也反复断句了几次，不过这个在加载用户进程时咱们再细说，在此我们只需要完成内存空间划分就行了。

以上我们讨论的结果是：虚拟地址空间的 0～3GB 是用户进程，3GB～4GB 是操作系统。下一节我们将聊聊具体实现。

5.2.5　启用分页机制

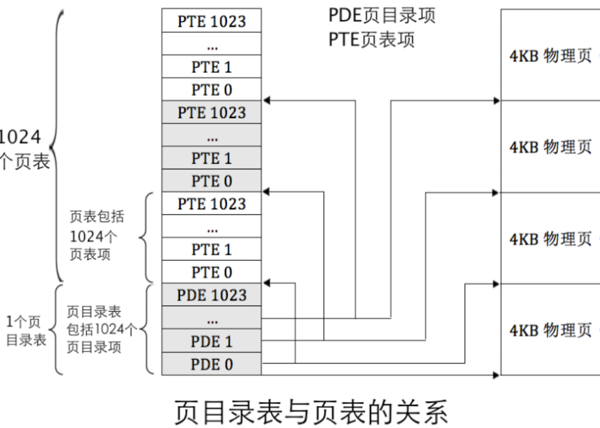
我们即将打开分页机制，从此我们的程序将在虚拟地址空间中运行。有了前面的理论知识，说干就干，咱们实践一把看看。

在实践之前，我们的脑子中得有个页表的印象，我们页表将按照如下方式部署，如图 5-20 所示。

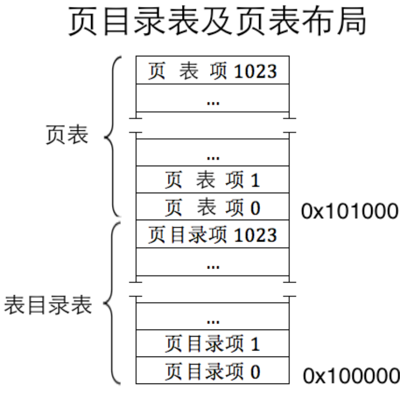
分页机制得有页目录表，页目录表中的是页目录项，其中记录的是页表的物理地址及相关属性，所以还得有页表。我们实际的页目录表及页表也将按照此空间位置部署，地址的最下面是页目录表，往上依次是页表。

页目录表和页表都存在于物理内存之中，它总该有个“安身”的地址。我们把它们安装在哪里呢？

页目录表的位置，我们就放在物理地址 0x100000 处。为了让页表和页目录表紧凑一些（这不是必须的），咱们让页表紧挨着页目录表。页目录本身占 4KB，所以第一个页表的物理地址是 0x101000。它们的物理布局如图 5-21 所示。



▲图 5-20　页目录表与页表的关系



▲图 5-21　页目录表与页表内存布局

首先关于分页功能的全部代码是代码 5-3，并不是下面的代码 5-2，代码 5-2 只是在代码 5-3 中调用的函数，它用于建立页目录表和页表，我想先把它给大家交待清楚，知道了实际内存安排后有利于理解后面分页机制的其他部分。如果大伙儿心急的话，可以先看看代码 5-3，从全局上看看代码流程。

代码 5-2 是为了完成启用内存分页机制三步曲之第一步：准备好页目录及页表。其中原委下面听小弟慢慢说来。

代码 5-2　（project/c5/b/boot/loader.S）

```
181 ;-------------   创建页目录及页表   ---------------
182 setup_page:
183 ;先把页目录占用的空间逐字节清 0
184     mov ecx, 4096
185     mov esi, 0
186 .clear_page_dir:
187     mov byte [PAGE_DIR_TABLE_POS + esi], 0
188     inc esi
189     loop .clear_page_dir
190
191 ;开始创建页目录项(PDE)
192 .create_pde: ; 创建 Page Directory Entry
193     mov eax, PAGE_DIR_TABLE_POS
194     add eax, 0x1000        ; 此时 eax 为第一个页表的位置及属性
195     mov ebx, eax   ; 此处为 ebx 赋值，是为.create_pte 做准备，ebx 为基址
196
197 ;   下面将页目录项 0 和 0xc00 都存为第一个页表的地址，每个页表表示 4MB 内存
198 ;   这样 0xc03fffff 以下的地址和 0x003fffff 以下的地址都指向相同的页表
199 ;   这是为将地址映射为内核地址做准备
200     or eax, PG_US_U | PG_RW_W | PG_P
      ; 页目录项的属性 RW 和 P 位为 1，US 为 1，表示用户属性，所有特权级别都可以访问
201     mov [PAGE_DIR_TABLE_POS + 0x0], eax      ; 第 1 个目录项
            ;在页目录表中的第 1 个目录项写入第一个页表的位置(0x101000)及属性(7)

202     mov [PAGE_DIR_TABLE_POS + 0xc00], eax
; 一个页表项占用 4 字节
          ; 0xc00 表示第 768 个页表占用的目录项，0xc00 以上的目录项用于内核空间
203        ;也就是页表的 0xc0000000～0xffffffff 共计 1G 属于内核
          ; 0x0～0xbfffffff 共计 3G 属于用户进程
204     sub eax, 0x1000
205     mov [PAGE_DIR_TABLE_POS + 4092], eax
; 使最后一个目录项指向页目录表自己的地址
206
207 ;下面创建页表项(PTE)
208     mov ecx, 256                   ; 1M 低端内存 / 每页大小 4k = 256
209     mov esi, 0
210     mov edx, PG_US_U | PG_RW_W | PG_P     ; 属性为 7，US=1，RW=1，P=1
211 .create_pte:     ; 创建 Page Table Entry
212     mov [ebx+esi*4],edx
; 此时的 ebx 已经在上面通过 eax 赋值为 0x101000，也就是第一个页表的地址

213     add edx,4096
214     inc esi
215     loop .create_pte
216
217 ;创建内核其他页表的 PDE
218     mov eax, PAGE_DIR_TABLE_POS
219     add eax, 0x2000                 ; 此时 eax 为第二个页表的位置
220     or eax, PG_US_U | PG_RW_W | PG_P  ; 页目录项的属性 US、RW 和 P 位都为 1
221     mov ebx, PAGE_DIR_TABLE_POS
222     mov ecx, 254                   ; 范围为第 769～1022 的所有目录项数量
223     mov esi, 769
224 .create_kernel_pde:
225     mov [ebx+esi*4], eax
226     inc esi
227     add eax, 0x1000
228     loop .create_kernel_pde
229     ret
```

先跟大家笼统地说下代码 5-2 的工作，大致上分成两部分，第一部分先建立个页目录表，后面的第二部分建立个页表。

由于我们的系统不是像 Linux 那样的庞然大物，现在啥都讲究个“定位”，咱们的定位就是学习操作系统的原理以及简单的实现，所以，代码量一定要小，这样大家才有可能坚持下来。目测完成之后，内核体积大概就是 70KB 以内，所以咱们还是充分利用低端 1MB 内存，以后会把内核加载到低端 1MB 之内，就在这一亩三分地上折腾就行啦。当然，依然可以将其放到 1MB 之上的任意位置，看您自己的安排啦，在本书中，咱们的 mbr、loader、操作系统内核都将放置在这 1MB 空间内，当然这 1MB 是指物理内存的 0～0xfffff。

交待完了，大家有没有疑问？比如我刚刚说的内核在物理内存 1MB 之内，但之前又说将内核地址映射到虚拟地址 3GB 之上。这如何做到？也许有同学马上已经抢答了，对喽，就是将虚拟地址 0xc0000000 之上的 1MB 地址映射到物理内存 1MB 之内。

正式工作开始啦。

第 181～189 行先清空页目录表所占的内存，由于内存中有大量随机数据，还是小心一点好，咱们先将它们统一初始化为 0。我这个人太小心啦，采取的是逐字节清 0 的方式。页目录表大小是 4KB，也就是 4096 个字节，这里用 loop 循环指令逐个清 0。loop 指令会用到 ecx 做循环计数器，所以为 ecx 寄存器赋值为 4096，每次循环一次后，ecx 会减 1，直到 ecx 为 0。值得说的是 PAGE_DIR_TABLE_POS 这个宏，它定义在 include.inc 中，这个宏用于定义页目录表的物理地址。那我们现在插播个小插曲介绍下新的属性，见代码 5-3，其中列出了部分新增内容。

代码 5-3　（project/c5/b/boot/include/boot.inc）

```
  1 ;-------------   loader和kernel   ----------
……
  9 PAGE_DIR_TABLE_POS equ 0x100000
……
 43 ;----------------   页表相关属性    --------------
 44 PG_P  equ   1b
 45 PG_RW_R  equ  00b
 46 PG_RW_W equ  10b
 47 PG_US_S  equ  000b
 48 PG_US_U  equ  100b
```

代码 5-3 中的第 9 行 PAGE_DIR_TABLE_POS 用于定义页目录表的物理地址，它的值为 0x100000，也就是我们会把页目录表放置到物理内存 0x100000。这是出了低端 1MB 空间的第一个字节。第 43～48 行是用于页目录项 PDE 和页表项 PTE 中的属性，是用二进制直接定义的，因此各二进制数字都以字符 b 结尾。拿 PG_US_S 举例，它表示 PTE 和 PDE 的 US 属性值为 S，这里把 S 的值定义为 000b，注意，虽然写了 3 个 0，最右边的两个 0 是占位的，只有最左边的 0 才表示 US 的值，因此 US 位的值为 0，这表示此 PTE 或 PDE 指向的内存不能被特权级 3 的任务访问，处理器只允许特权级为 0、1、2 的任务访问该 PTE 或 PDE 指向的内存。您懂的，PG_US_U 的意思是 US 位的值为 1，处理器允许任何特权级的任务访问 PTE 或 PDE 指向的内存。PG_P 表示 PTE 或 PDE 指向的物理内存页框已位于内存中，这个 P 位的作用已经和大伙儿介绍过了，当物理内存不足时，操作系统的虚拟内存管理机制有可能会将该 PDE 或 PTE 指向的物理页框换出到磁盘上，此时 PDE 或 PTE 的 P 位便被操作系统置为 0，处理器访问该 PDE 或 PTE 时会触发 page_fault 缺页异常，操作系统为该异常注册了中断处理程序，该程序会将所缺的页从磁盘上重新加载到内存中，并将 P 位置为 1。中断处理程序运行结束后，处理器会再次该 PDE 或 PTE，发现 P 位为 1，顺利通过。PG_RW_W 和 PG_RW_R 分别表示 PDE 或 PTE 指向的物理内存可写、只读。好啦，插曲结束啦，赶紧回到代码 5-2。

代码 5-2 的第 187 行，是利用 PAGE_DIR_TABLE_POS 作为基址，esi 作为变址，然后通过 188 行的 inc esi，每次使 esi 自增 1，逐步完成 4096 字节的清 0 工作。

第 191～205 行是创建页目录项，由于代码中的注释已经很详尽了，这里不再一一陈述。重点说一下第 197～205 行。寄存器 eax 是页目录项的内容（提醒一下，PG_US_U | PG_RW_W | PG_P 逻辑或的结果是 0x7），分别将其存入到页目录项的第 0 项和第 768 项，0xc00/4=768。页目录项代表一个页表，也就是说，这两处都是指向同一个页表。首先明确一下，它们共同所指向的这个页表地址是 0x101000，它将来要指向的物理地址范围是 0～0xfffff，只是 1MB 的空间，其余 3MB 并未分配。这是我们设计好的。一会儿在说创建页表时就知道啦。

为什么要在两处指向同一个页表？原因是我们在加载内核之前，程序中运行的一直都是 loader，它本身的代码都是在 1MB 之内，必须保证之前段机制下的线性地址和分页后的虚拟地址对应的物理地址一致。第 0 个页目录项代表的页表，其表示的空间是 0～0x3fffff，包括了 1MB（0～0xfffff），所以用了第 0 项来保证 loader 在分页机制下依然运行正确。那为什么也要把该地址放置到第 768 项呢？前面说过啦，我们将来会把操作系统内核放在低端 1M 物理内存空间，但操作系统的虚拟地址是 0xc0000000 以上，该虚拟地址对应的页目录项是第 768 个。这个算起来容易，0xc0000000 的高 10 位是 0x300，即十进制的 768。这样虚拟地址 0xc0000000～0xc03fffff 之间的内存都指向的是低端 4MB 之内的物理地址，这自然包括操作系统所占的低端 1MB 物理内存。从而实现了操作系统高 3GB 以上的虚拟地址对应到了低端 1MB，也就是如前所说我们内核所占的就是低端 1MB。

第 204～205 行的目的是在页目录的最后一个页目录项中写入页表自己的物理地址。eax 原来的值是 0x101007，这是第一个页表的页目录项。将 eax 减去 0x1000 后，eax 的值为 0x100007。也许您在想，为什么使用属性 PG_US_U，而不是 PG_US_S？原因是这样的，PG_US_U 和 PG_US_S 是 PDE 或 PTE 的属性，它用来限制某些特权级的任务对此内存空间的访问（无论该内存空间中存放的是指令，还是普通数据）。PG_US_U 表示 PDE 或 PTE 的 US 位为 1，这说明处理器允许任意 4 个特权级的任务都可以访问此 PDE 或 PDE 指向的内存。PG_US_S 表示 PDE 或 PTE 的 US 位为 0，这说明处理器允许除特权级 3 外的其他特权级任务访问此 PDE 或 PDE 指向的内存。此时若使用属性 PG_US_S 也没问题，不过将来我们会实现 init 进程，它是用户级程序，而它位于内核地址空间，也就是说将来我们会在特权级 3 的情况下执行 init，这会访问到内核空间，这就是此处用属性 PG_US_U 的目的，好啦，这就解释到这，我们继续。eax 本身是页目录项，现在将其加入到页目录表中最后一个页目录项中，目的是为了将来能够动态操作页表。在这里大伙儿先有这么个概念，在讲述完代码后咱们再细说。

第 207～215 行是创建页表，我们前面所说的“设计好的”，就是指这个页表。此页表是页目录中的第 0 个页目录项对应的页表，它用来分配物理地址范围 0～0x3fffff 之间的物理页，这也就是虚拟地址 0x0～0x3fffff 和虚拟地址 0xc0000000～0xc03fffff 对应的物理页。一个页表表示的内存容量是 4MB，但我们目前只用到了第 1 个 1MB 空间，所以我们只为这 1MB 空间对应的页表项分配物理页。每个物理页是 4KB，所以只需要 1MB/4KB=256 个页表项即可。同样是用 loop 指令循环为页表项赋值，所以 ecx 作为循环计数器被赋值为 256。

大家可以看出，第 213 行的“add edx，4096”，edx 是物理页的页表项，每次 edx 加上 4KB，其物理地址是连续分配的，即在低端 1MB 内存中，虚拟地址等于物理地址。

第 217～228 行创建除第 768 个页表之外的其他页表对应的 PDE，也就是内核空间中除第 0 个页表外的其余所有页表对应的目录项。虽然前面已经创建了第 768 个页表的 PDE 了，但它只是一个页表的空间。尽管我们的内核甚至连 4MB 的内存空间都绰绰有余，也就是说 1 个页表足矣应付了，只需要在页目录表中安装一个 PDE 就够了，但为了真正实现内核被所有进程共享，还是在页目录表中为内核额外安装了 254 个页表的 PDE（第 255 个 PDE 已经指向了页目录表本身），也就是内核空间的实际大小是 1GB 减去 4MB 的差，当然了，必须要为页表中具体的 PTE 分配物理页框后才算真正的内存空间，此处还不算，此处在页目录表中把内核空间的目录项写满，目的是为将来的用户进程做准备，使所有用户进程共享内核空间。这方面内容有些超前了，但还是给大伙儿提前解释下，我们将来要完成的任务是让每个用户进程都有独立的页表，也就是独立的虚拟 4GB 空间。其中低 3GB 属于用户进程自己的空间，高 1GB 是内核空间，内核将被所有用户进程共享。为了实现所有用户进程共享内核，各用户进程的高 1GB 必须“都”指向内核所在的物理内存空间，也就是说每个进程页目录表中第 768～1022 个页目录项都是与其他进程相同的（各进程页目录表中第 1023 个目录项指向页目录表自身），因此在为用户进程创建页表时，我们应该把内核页表中第 768～1022 个页目录项复制到用户进程页目录表中的相同位置。一个页目录项对应一个页表地址，页表地址固定了，后来新增的页表项也只会加在这些固定的页表中。如果不这样的话，进程陷入内核时，假设内核为了某些需求为内核空间新增页表（通常是申请大量内存），因此还需要把新内核页表同步到其他进程的页表中，否则内核无法被“完全”共享，只能是“部分”共享。所以，实现内核完全共享最简单的办法是提前把内核的所有页目录项定下来，也就是提前把内核的页表固定下来，这是实现内核共享的关

键。这样一来，内核所在的空间完全被所有进程共享，所有进程都可以使用内核提供的服务，内核若为任意一个用户进程在内核空间中创建了某些资源的话，其他进程都可以访问到该资源。由于我们尚未接触到用户进程方面的内容，怕您越看越晕乎，让我们具体一点说，如果此处仅仅是安装第 768 个页目录项的话，第 769～1022 个目录项是空的，将来即使把第 768～1022 个目录项复制给用户进程时，内核空间也仅是其低 4MB 被所有进程共享，万一在某些情况下内核使用的空间超过 4MB，要用到第 769 个页目录项对应的页表，由于此处未提前准备该目录项，创建的新页表（的 PDE）只会安装在当时那个进程的页目录表中，而切换了其他进程后，新进程的页目录表中并不包含新页表的 PDE，因此无法访问到最新的内核空间。如果您觉得晕乎也没关系，毕竟这涉及了一些超前的内容，以后到实现用户进程的时候您就会了解啦，好啦，原理介绍过了，代码部分还是很简单的，同创建 pte 的思路类似，不多说了。

第 229 行是通过 ret 指令返回，自此函数 setup_page 结束。

说完了 setup_page 函数后，我们可以正式演奏启用分页的三步曲啦。跟建页表相比，这个过程比较容易，我们马上就要进入分页模式啦。请看代码 5-4。

代码 5-4　（project/c5/b/boot/loader.S）

```
149  ; 创建页目录及页表并初始化页内存位图
150  call setup_page
151
152  ;要将描述符表地址及偏移量写入内存 gdt_ptr，一会儿用新地址重新加载
153  sgdt [gdt_ptr]            ; 存储到原来 gdt 所有的位置
154
155  ;将 gdt 描述符中视频段描述符中的段基址+0xc0000000
156  mov ebx, [gdt_ptr + 2]
157  or dword [ebx + 0x18 + 4], 0xc0000000
    ;视频段是第 3 个段描述符，每个描述符是 8 字节，故 0x18
158  ;段描述符的高 4 字节的最高位是段基址的第 31～24 位
159
160  ;将 gdt 的基址加上 0xc0000000 使其成为内核所在的高地址
161  add dword [gdt_ptr + 2], 0xc0000000
162
163  add esp, 0xc0000000       ; 将栈指针同样映射到内核地址
164
165  ; 把页目录地址赋给 cr3
166  mov eax, PAGE_DIR_TABLE_POS
167  mov cr3, eax
168
169  ; 打开 cr0 的 pg 位（第 31 位）
170  mov eax, cr0
171  or eax, 0x80000000
172  mov cr0, eax
173
174  ;在开启分页后，用 gdt 新的地址重新加载
175  lgdt [gdt_ptr]            ; 重新加载
176
177  mov byte [gs:160], 'V'
;视频段段基址已经被更新，用字符 v 表示 virtual addr
178
179  jmp $
```

代码 5-4 是启用分页的全部代码，也就是完成了分页机制的三步曲。

第 150 行先建立页目录表和所需要的页表，这就是刚才咱们所说的代码 5-2 中的函数，开启分页的第一步：先准备好页表。

第 152～153 行，是为了重启加载 GDT 做准备。因为我们在页表中会将内核放置到 3GB 以上的地址，我们也把 GDT 放在内核的地址空间，在此通过 sgdt 指令，将 GDT 的起始地址和偏移量信息 dump（像倒水一样）出来，依然存放到 gdt_ptr 处，一会儿待条件成熟时，我们再从地址 gdt_ptr 处重新加载 GDT。

第 155～158 行是修改显存段的段描述符的段基址，因为将来内核运行在 3GB 以上，打印功能将来也是在内核中实现，肯定不能让用户进程直接能控制显存。故显存段的段基址也要改为 3GB 以上才行。大家都知道 32 位虚拟地址空间共 4GB，若用十六进制表示，最高位（第 31 位）每变化 4 位就表示 1GB 空间，也没什么高深的，其实就是 16 位/4GB=4 位/1GB，意为每 1GB 内存空间需要 4 位来表示。所以，

0x00000000～0x3fffffff 是第 1 个 1GB 内存，0x40000000～0x7fffffff 是第 2 个 1GB。0x80000000～0xbfffffff 是第 3 个 1GB 内存，0xc0000000～0xffffffff 是第 4 个 1GB 内存。内核是 3GB 以上，范围就是第 4 个 1GB，故虚拟地址 0xc0000000～0xffffffff 才是内核地址。

gdt_ptr 处的值包括两部分，前部分是 2 字节大小的偏移量，其后是 4 字节大小 GDT 基址。这里先要得到 GDT 地址，所以在 156 行 gdt_ptr 加了 2，即“mov ebx, [gdt_ptr + 2]”。之后 ebx 是 GDT 的地址。由于显存段描述符是 GDT 中第 3 个描述符，一个描述符大小是 8 字节，所以 ebx 要加上 0x18 才能访问显存段描述符。这里要将段描述符的基地址修改为 3GB 以上，所以在原有地址的基础上要加上 0xc0000000。大伙儿观察这个数，只有最高位是 c，其他位都为 0，段描述符中记录段基址最高位的部分是在段描述符的高 4 字节的最高 1 字节，所以 ebx 不仅要加上 0x18，还要加上 0x4。为了省事，我们直接将整个 4 字节做或运算。最后就是第 157 行的指令“or dword [ebx + 0x18 + 4], 0xc0000000”。

在修改完了显存段描述符后，现在可以修改 GDT 基址啦，我们把 GDT 也移到内核空间中。所以第 161 行，直接将 gdt_ptr+2 处的 GDT 基址加了 0xc0000000。其实这不是必须的，如果分页后不重复加载 GDT 的话，也可以不修改 GDT 基址。

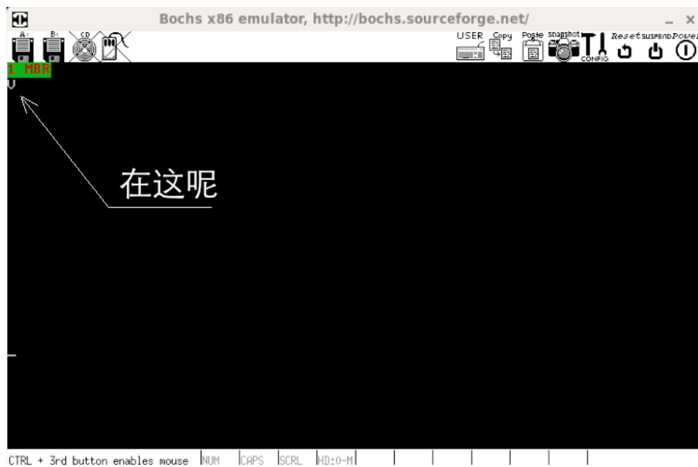
大伙儿千万不要忘记把栈地址也修改成内核使用的地址，所以 163 行直接把 esp 加了 0xc0000000。

第 165～172 行是启用分页机制的第二步及第三步，将页目录表物理地址赋值给 cr3 寄存器后，随后启用 cr0 寄存器的 pg 位。

在一切妥妥的之后，在第 174 行把 GDT 重新加载啦。

为了检查在分页机制下我们的程序是否工作正常，在第 177 行直接在新的显存地址中写入字符 V，“mov byte [gs:160], 'V'”中的 160 是指第二行开始的位置，一个字符是 2 字节，每行是 80 个字符，你懂的。

好啦，代码是说完了，必须要运行一下看看啦。运行效果如图 5-22 所示。



▲图 5-22　分页模式下的运行效果

在分页后，GDT 的基址会变成 3GB 之上的虚拟地址，显存段基址也变成了 3GB 这上的虚拟地址，如图 5-23 所示。

图 5-23 中上面的框框是新的 GDT 的段基址，已经变成了 0xc0000900，下面的框框是显存段描述符的段基址，已经是 0xc00b8000，不再是 0xb8000 了。

```
<bochs:3> info gdt
Global Descriptor Table (base=0xc0000900, limit=31):
GDT[0x00]=??? descriptor hi=0x00000000, lo=0x00000000
GDT[0x01]=Code segment, base=0x00000000, limit=0xffffffff, Execute-Only, Non-Con
forming, Accessed, 32-bit
GDT[0x02]=Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed
GDT[0x03]=Data segment, base=0xc00b8000, limit=0x00008fff, Read/Write, Accessed
You can list individual entries with 'info gdt [NUM]' or groups with 'info gdt
NUM] [NUM]'
<bochs:4>
```

▲图 5-23　分页后 GDT 的变化

5.2.6　用虚拟地址访问页表

我们已经进入了分页机制运行模式，数据在内存中以物理地址来寻址，最终是在物理地址上进行 I/O 操作，这意味着，以后我们访问任何物理地址都需要通过虚拟地址进行啦。

页表是一种动态的数据结构，有时候需要给页表“增肥”，比如申请一块内存时，需要往里面添加页表项或者页目录项。有时候也需要为页表“减肥”，比如在释放一块内存时，页表中相应的页表项或页目录项都要清零。这正是二级页表灵活的地方，根据需要动态增减。

可是，页表也位于内存中，修改页表的操作也需要通过内存地址才行。而且我们说过啦，页表是将虚拟地址转换成物理地址的映射表，在分页机制下，如何用虚拟地址访问到页表自身呢？

其实有两种方式，一种方式是最简单的，让虚拟地址直接与物理地址一一对应。就像咱们低端 1MB 的虚拟内存空间一样，其与物理内存 1MB 是一一对应的，在 0～1MB 之间，访问其中任何一个虚拟地址，最终都转换成与其等值的物理地址。我们可以让虚拟地址的 0～4GB 与物理地址的 0～4GB 一一对应起来，这样，访问哪块物理地址，直接访问其虚拟地址就好了。这么做的优点是简单直接，但缺点是并不能很好地体现虚拟地址的优势：虚拟地址可以与任何一个物理地址对应，这种乱序的映射关系才是虚拟地址的魅力所在。我们既然是来学习操作系统原理的，那我们就不偷懒了，一切以学习为目的，所以我们采用另一种方式，让虚拟地址与物理地址乱序映射。

其实在前面讲述分页机制时，我们就已经为这种方式埋下了伏笔，还记得我们在最后一个页目录项中填入了页目录表的物理地址吗？就是代码 5-2 第 205 行的“mov [PAGE_DIR_TABLE_POS + 4092]， eax”，这句代码就是用虚拟地址访问页表的关键步骤。虽然我们为此占用了一个页目录项，也就是少了 4KB 的内存空间，但我们占用的是最顶端的 4KB，很少有程序能将 4GB 虚拟内存空间全部占满，我们姑且先抱着侥幸的心理进行，一般情况不会出现问题。

先得做到通过虚拟地址访问页表，才能使通过虚拟地址修改页表成为可能。下面咱们先从虚拟地址入手，找到能访问页表的“虚拟入口”。由于大家刚刚接触到分页，所以咱们循序渐进，先从“正常”的虚拟地址开始。

开启分页之后，除了可以在物理内存 0x100000 处后看到页目录表外，我们还可以在虚拟机中利用 info tab 命令看到页表中虚拟地址到物理地址的映射关系。info 是用来查看各种数据的命令，tab 表示页表。图 5-24 所示是咱们目前的虚拟地址映射情况。

```
<bochs:4> info tab
cr3: 0x000000100000
0x00000000-0x000fffff -> 0x000000000000-0x0000000fffff
0xc0000000-0xc00fffff -> 0x000000000000-0x0000000fffff
0xffc00000-0xffc00fff -> 0x000000101000-0x000000101fff
0xfff00000-0xfff00fff -> 0x000000101000-0x000000101fff
0xfffff000-0xffffffff -> 0x000000100000-0x000000100fff
<bochs:5>
```

▲图 5-24　虚拟地址与物理地址映射

图 5-24 中，在键入 info tab 命令后，cr3 寄存器显示的是页目录表的物理地址。按->分成左右两列，左边列出的是 32 位虚拟地址范围，右边是虚拟地址对应的物理地址，不过其用 48 位来表示，这个我也不知道是为什么，可能传说中的 64 位扩展，不过这个不重要。现在咱们分析下各行的映射结果。

看第一行，虚拟地址 0x00000000～0x000fffff，这是虚拟空间低端 1M 内存，其对应的物理地址是 0x000000000000～0x0000000fffff。这是咱们的第 0 个页表起的作用，可以翻翻上面的代码 5-2，就是 ecx=256 的那个，为 256 个页表项分配物理页。

第二行，虚拟地址 0xc0000000～0xc00fffff，这是咱们第 768 个页表起的作用。由于第 0 个页目录项和第 768 个页目录项指向的是同一个页表，所以其映射的物理地址依然是 0x000000000000～0x0000000fffff。这和咱们之前的安排是一致的。

以上这两组线性地址到物理地址的映射，看上去比较正常，这毕竟是咱们有意安排的。但下面这三行映射就显得比较怪异了。

0xffc00000～0xffc00fff -> 0x000000101000～0x000000101fff

0xfff00000～0xfff00fff -> 0x000000101000～0x000000101fff

0xfffff000～0xffffffff -> 0x000000100000～0x000000100fff

现在是兑现承诺的时候了，还记得之前将咱们将页目录表的地址写入了最后一个页目录项，就是代码5-2 第 205 行的“mov [PAGE_DIR_TABLE_POS + 4092]， eax”，由于它的作用，我们在 info tab 的输出中，才会看到了这三个怪异的虚拟地址。咱们分析一下，这三个奇怪的虚拟地址是怎么来的。

第一个奇怪的虚拟地址映射：

0xffc00000～0xffc00fff -> 0x000000101000～0x000000101fff

最后一个目录项是第 1023 个目录项，我们已经知道，虚拟地址的高 10 位用来访问页目录表中的目录项，如果高 10 全为 1 ，即 1111111111b=0x3ff=1023，则访问的是最后一个目录项，该目录项中的高 20 位是页目录表本身的物理地址 0x100000。不过，由于该地址是经过虚拟地址的高 10 位索引到的，所以被认为是个页表的地址，也就是说，页目录表此时被当作页表来用啦。线性地址的中间 10 位用来在页表中定位一个页表项，从该页表项中获取物理页地址。这时，若虚拟地址中间 10 位为 0000000000b=0x0，即检索到第 0 个页表项，此时的页目录表作为页表，故第 0 个页表项实际上是页目录表的第 0 个页目录项，其中记录的是第一个页表的物理地址，其值是 0x101000，此值被认为是最终的物理页地址。如果虚拟地址的低 12 位为 000000000000b=0x000，最终的物理地址是 0x101000+0x000=0x101000。故虚拟地址是 0xffc00000～0xffc00fff ，其被映射的物理地址范围是 0x00101000～0x00101fff。第一个奇怪的地址映射说通了。

提取出关键一点：高 10 位若为 0x3ff，则会访问到页目录表中最后一个页目录项，由于页表中也是 1024 个页表项，故中间 10 位若为 0x3ff，则会访问到页表中最后一个页表项。

第二个奇怪的虚拟地址映射：

0xfff00000～0xfff00fff -> 0x000000101000～0x000000101fff

虚拟地址 0xfff00000 的高 10 位依然为 0x3ff，中间 10 位是 1100000000b=0x300，这是第 768 个页目录项，该页目录项指向的页表与第 0 个页目录项指向的页表相同。所以虚拟地址 0xfff00000 映射为物理地址 0x00101000 成立，这下大家也容易理解 0xfff00fff 映射为 0x00101fff。

第三个奇怪的虚拟地址映射：

0xfffff000～0xffffffff -> 0x000000100000～0x000000100fff

0xfffff000 的高 10 位是 0x3ff，中间 10 位依然是 0x3ff，大家将其换成二进制后就容易看出来了。如果高 10 位为 0x3ff，则会访问到最后一个页目录项，该页目录项中是我们的页目录表的物理地址。目录项中的应该是页表的物理地址，所以此页目录表被当作页表来用。中间 10 位也是 0x3ff，用它在页表内索引页表项，在此是在页目录表中索引，所以，索引到的是最后一个项目，页部件认为该项是页表项，但其实是页目录项，该页目录项中的是页目录表的物理地址。虚拟地址的低 12 位是 0x000，所以得到的物理地址最终是页目录表的物理地址+0x000=页目录表的物理地址。我们的页目录表物理地址是 0x00100000。于是虚拟地址 0xfffff000 映射成为物理地址 0x00100000 成立，也同样容易理解 0xffffffff 映射为 0x00100fff。

提炼再提炼，如果虚拟地址的高 20 位为 0xfffff，经过我们的页目录表映射，将会访问到页目录表自己的物理地址。

基于这一点，我们在访问页目录表中的页目录项时，可以通过虚拟地址 0xfffffxxx 的方式，其中的 xxx 是页目录表内的偏移地址，由于它是 12 位的，已经可以用来表示 4KB 的范围了，所以并不会被页部件当作目录项或页表项的索引，也就是并不会被页部件自动乘以 4。故 xxx 是以字节为单位的偏移，是已经由程序员自己手工乘以 4 后的值。

总结一下用虚拟地址获取页表中各数据类型的方法。

- 获取页目录表物理地址：让虚拟地址的高 20 位为 0xfffff，低 12 位为 0x000，即 0xfffff000，这也是页目录表中第 0 个页目录项自身的物理地址。
- 访问页目录中的页目录项，即获取页表物理地址：要使虚拟地址为 0xfffffxxx，其中 xxx 是页目录项的索引乘以 4 的积。

- 访问页表中的页表项：要使虚拟地址高 10 位为 0x3ff，目的是获取页目录表物理地址。中间 10 位为页表的索引，因为是 10 位的索引值，所以这里不用乘以 4。低 12 位为页表内的偏移地址，用来定位页表项，它必须是已经乘以 4 后的值。

公式为 0x3ff<<22+中间 10 位<<12+低 12 位。

5.2.7　快表 TLB（Translation Lookaside Buffer）简介

分页机制虽然很灵活，但您也看到了，为了实现虚拟地址到物理地址的映射，过程还是有些麻烦的。先要从 CR3 寄存器中获取页目录表物理地址，然后用虚拟地址的高 10 位乘以 4 的积作为在页目录表中的偏移量去寻址目录项 pde，从 pde 中读出页表物理地址，然后再用虚拟地址的中间 10 位乘以 4 的积作为在该页表中的偏移量去寻址页表项 pte，从该 pte 中读出页框物理地址，用虚拟地址的低 12 位作为该物理页框的偏移量，呼……终于完成虚拟地址到物理地址的映射。

每一个虚拟地址到物理地址的转换都要重复以上过程，甭说真正去做了，光描述这个过程我都觉得繁琐，何况这只是用二级页表做地址映射的过程，要是用三级页表……我都替处理器喊累。不只如此，处理器的速度和内存的速度完全是两个数量级，页表毕竟在内存中，转换过程中频繁的内存访问，使得地址转换速度慢上加慢，而处理器也不得不停下来等待内存的响应。

虚拟地址到物理地址的转换，最终是想得到虚拟地址所对应的物理地址，如果给出一个虚拟地址后能直接得到相应的页框物理地址，免去中间的查表过程，直接用虚拟地址的低 12 位在该物理页框中寻址，岂不是大大提高了地址转换速度。根据程序的局部性原理，可以将近来常用的地址和指令加载到速度更快的设备中，因此我们都想到了缓存。处理器准备了一个高速缓存，可以匹配高速的处理器速率和低速的内存访问速度，它专门用来存放虚拟地址页框与物理地址页框的映射关系，这个调整缓存就是 TLB，即 Translation Lookaside Buffer，俗称快表，其结构如图 5-25 所示。

虚拟地址的高 20 位（虚拟页框号）	属性	物理地址的高 20 位（物理页框号）
...		
...		

▲图 5-25　TLB 结构简图

TLB 中的条目是虚拟地址的高 20 位到物理地址高 20 位的映射结果，实际上就是从虚拟页框到物理页框的映射。除此之外 TLB 中还有一些属性位，比如页表项的 RW 属性。

有了 TLB，处理器在寻址之前会用虚拟地址的高 20 位作为索引来查找 TLB 中的相关条目，如果命中（匹配到相关条目）则返回虚拟地址所映射的物理页框地址，否则会查询内存中的页表，获得页框物理地址后再更新 TLB。

高速缓存由于成本等原因，容量一般都很小，TLB 也是，因此 TLB 中的数据只是当前任务的部分页表，而且只有 P 位为 1 的页表项才有资格在 TLB 中，如果 TLB 被装满了，需要将很少使用的条目换出。

缓存相当于数据源的快照，为了保证缓存与数据源同步变化，这就涉及到缓存刷新的问题。TLB 也是缓存，当内存中的原页表被修改时，TLB 中的相应映射关系按理说也要更新。一般的缓存可以定期刷新，甚至推迟几分钟都可以，但 TLB 和一般的缓存可不一样，你想，TLB 是页表的缓存，处理器寻址时最先访问的是 TLB，TLB 里面存储的是程序运行所依赖的指令和数据的内存地址，任意时刻都必须保证地址的有效性，否则程序必然出错，所以 TLB 必须实时更新。可是如果实时读取内存中的页表去更新 TLB 的话，这又回到了从内存查询映射关系的老路，TLB 反而成了鸡肋。为此，TLB 并不自动更新，处理器也不负责 TLB 的有效性，它把 TLB 的维护工作交给操作系统开发人员，由开发人员手动控制。这的确是非常合理的，毕竟维护页表的代码是开发人员自己写的，他们肯定知道何时修改了页表，或是修改了哪些条目。

尽管 TLB 对开发人员不可见，但依然有两种方法可以间接更新 TLB，一个是针对 TLB 中所有条目的方法——重新加载 CR3，比如将 CR3 寄存器的数据读出来后再写入 CR3，这会使整个 TLB 失效。另一个方法是针对 TLB 中某个条目的更新。处理器提供了指令 invlpg（invalidate page），它用于在 TLB 中刷新某个虚拟地址对应的条目，处理器是用虚拟地址来检索 TLB 的，因此很自然地，指令 invlpg 的操作数也是虚拟地址，其指令格式为 invlpg m。注意，其中 m 表示操作数为虚拟内存地址，并不是立即数，比如要更新虚拟地址 0x1234 对应的条目，指令为 invlpg [0x1234]，并不是 invlpg 0x1234。将来咱们在内存管理系统中会涉及到 TLB 的更新操作，这一点应注意。

有关 TLB 的介绍就到这儿，下节再见。

5.3 加载内核

经过了前面漫长的旅行，今天终于到了有关内核方面的内容，我想大家的心情一定是万分激动的，毕竟这才是真正的操作系统部分。

前面为了帮助大家理解相关内容，铺垫了太多的基础知识，以至于大家对此学习旅程有望不到边的感觉。我非常理解大家，但我也更相信磨刀不误砍柴工，对底层知识的充分掌握才能更好更快地理解上层应用的变化万千。

今天开始我们正式踏上操作系统的学习旅程。

5.3.1 用 C 语言写内核

在这之前，我们一直用汇编语言直接与机器对话，如果大家不知道这个世界上有高级语言的话，我想大家也不会觉得写汇编代码的过程很辛苦，哈哈，幸福确实是比较出来的。相对于汇编语言，用 C 语言写内核是非常爽的事，马上我们就要步入内核实践中啦，所以现在和大家聊聊 C 语言写内核的体会。

通常，我们写的代码都是直接编译成可执行文件，那是因为我们是在写用户程序，操作系统为咱们提供了很多便利，所以编译和链接一气呵成，不需要咱们单独再指定什么，编译器也和操作系统达成了诸多约定，默默在后面为咱们做了大量的工作，比如程序编译出来的虚拟起始地址通常是 0x8048000 左右。在有操作系统为咱们撑腰时，我们确实不需要关注这些与业务逻辑无关的东西，只要专注于自己的工作就好啦。可如今，我们要用 C 语言写脱离操作系统的程序，这回咱们就不能再这么省心了，必须要自己指定程序的入口地址。

另外，我们之前开发用户程序，有大量的标准库可以用，标准库一般是系统调用的封装，所以，表面上通过标准库访问系统资源，本质上是用系统调用来实现的。当然如果大伙儿愿意，在用户程序中也可以直接调用“系统调用”，在功能上这是允许的，因为中断描述符表中系统调用对应的中断描述符，它的权限是用户程序可以访问的，否则就无法实现系统调用啦。就拿 Linux 来说，它的系统调用是先往 eax 寄存器中写入系统调用号，然后通过 0x80 中断来实现的。我们可以用汇编语言写一个系统调用的代码，用 C 语言去调用它或者干脆直接在 C 语言中内嵌汇编代码。无论是采用哪种形式，汇编语言的部分都是诸如先用 mov eax，xx 的形式在 eax 寄存器中指定系统调用的功能号，然后紧跟着使用中断指令 int 0x80 来引发 0x80 中断，从而触发 0x80 对应的中断处理程序，由该中断处理程序根据 eax 的内容去执行相应的系统调用。虽然可以直接调用“系统调用”的功能，但不推荐这样做，毕竟标准库中为咱们考虑了很多优化策略，通过标准库访问系统资源比直接用系统调用效率更高。也许有同学不信这个邪，非要整出个效率更高的库，当然这是非常可能的，可是标准库考虑的不仅是效率，还有很多兼容规范在里面，所以它可能会为了规范而牺牲效率。除非为了某些个性化的应用咱们才去写自己的库，否则还是不要企图颠覆标准库啦。标准库可以说是由世界上成千上万的超级大脑完成的，以咱们个人之力去和全世界的极客拼脑细胞，这是不科学的，不如把精力放在其他方面，好啦，我知道话又说多啦，大伙见笑啦。

对于系统调用这些平时我们认为理所当然的功能，如今已经成为了咱们的奢望。首先咱们本身是在写操作系统，而不是用户程序，操作系统不应该再依赖于其他系统的功能，所以不能在咱们的程序（操作系统）中再调用宿主操作系统的系统调用功能。其次，同一时刻只能有一个操作系统在运行，咱们即使调用了 0x80 中断，中断描述符表里 0x80 对应的中断处理程序是咱们提供的，再也不存在宿主系统的代码，相当于咱们在调用自己的中断处理程序，而此时我们可能尚未准备好相应的中断处理程序。如果系统调用不能用，就更不能用 C 标准库啦，所以只能用 C 语言原生支持的语法结构。不过以后我们会在实现内核的过程中建立咱们自己的库，库中会通过咱们自己的系统调用实现某些功能。

以上多说了几句有关系统调用的实现，其实我是怕无法满足好奇心强的同学，很担心仅仅一句“在脱离操作系统下写程序不能使用系统调用”让更多的同学感到不解。如果我解释得还不够，咱们以后会在实现系统调用的时候有所了解。

以下 5 段文字是为照顾到开发经验较少的同学，如果您经验较丰富，可以直接跳到下面第六段。

也许有的同学喜欢用汇编语言来实现操作系统，觉得用汇编来写程序似乎更简单直接，可控性比较强，有种“一切尽在掌握”的感觉。而用 C 语言实现操作系统这件事，虽然轻松很多，但似乎隐约感觉到有些慌张。因为虽然 C 语言相对来说更接近于人的逻辑思维，但恰恰是这种优越性，给一些好学的同学带来了困扰，毕竟咱们是在写底层的软件，必须要随心所欲地控制 CPU，要时时刻刻知道 CPU 在干什么。而感觉上，C 语言不能直接控制 CPU，比如无法直接控制寄存器，这里面有太多的黑盒子，无法掌控的东西似乎有很多，不知道编译器在后面是怎么将我的逻辑思维转换成机器指令的。这种黑盒式的操作确实让人觉得神秘又不放心。

不同语言应用在不同的层级，各层级有不同的思维方式，C 语言运用在更高的层级上，它的一行代码相当于多行汇编语言代码，因此 C 语言的语法对于汇编语言来说类似于一种需求。汇编语言相对来说运用在较低层级上，它是为完成宏观需求的具体步骤，在程序语言层面，汇编语言可以认为是不能再细分的最基本的原子。应用不同层级的语言，我们只要运用那个层级的思维即可。C 语言和汇编语言的关系就像产品经理和开发人员那样，产品经理在设计一款产品时，只需要提出需求，这是站在“高层”上的开发，而开发人员要将需求转换为具体的代码，需要在微观上细化那些“高层”的需求，对于这款产品来说，无论是产品经理，还是开发人员，他们都在自己的层级上开发。一个是以需求为粒度做开发，另一个是以代码为粒度做开发，一个是在“高层上”思考提哪些需求，另一个是在“底层上”思考如何满足需求。

汇编指令与机器指令几乎是一对一的，即一名汇编代码只对应一句具体的机器码，不会有更多对应的选择，所以可以认为汇编指令就是机器指令。C 语言的编译过程是先将 C 语言代码转换成汇编代码，然后再将汇编代码转换成机器指令。所以用 C 语言写出来的程序，最终可以转换成对应的一句或多句汇编指令。它们的关系就好比出租车上的乘客和司机，乘客只要告诉司机想去哪里就行了，其他的工作由司机和车共同配合完成。比如乘客说要去北京大学南门，司机根据当前的位置计算相对路径，比如先开车直行 1 公里，在路口处左转，再直行 2 公里后右转弯就到达了北京大学南门。乘客要去北京大学南门的这个需求就相当于 C 语言代码，这是上层需求。司机相当于编译器，由它将客户需求转换成具体的实现步骤，比如转换成踩油门直行、左转方向盘拐弯、再踩油门直行、再右转方向盘拐弯这四个驾驶操作，当然，司机只是发号施令，并不是司机在跑，真正把乘客带到目的地的工作者是出租车。出租车相当于 CPU，由它最终落实司机的驾驶操作，将乘客带到目的地，司机的这些驾驶操作相当于机器指令。站在乘客的角度，它只是说了一句话，就让汽车做了加油门、转弯等多个微操作，这就是 C 和机器指令之间的关系。

不知道我这样举例子，是否打消了您的疑虑，总之我们用 C 语言写程序，一定要充分相信 C 编译器的工作。

也许有人曾经想过，写操作系统已经是底层的事了，做底层的事就应该用更底层的东西来实现，必须用汇编语言或比汇编语言还要低层的东西。这种想法我非常理解，我学习之初也曾有过类似的猜想。当然，确实可以用更原始的东西来实现操作系统，但那样也更麻烦，需要极大的耐心和良好的体格，哈哈。语言只是个工具，对机器而言，它能接受的是机器指令，只要最终交给机器的是机器指令就成了。而 C 语言这种高级语言是可以被编译成机器指令的，就是我们平时编译出来的二进制文件，它里面都是二进制的机器指令，CPU 处理起来完全没有问题。选择哪种语言，只是实现的途径不同，最终还是汇总到机器指令那里。就像吃饭用筷子还是用勺子一样，饭最终还是被送到了嘴里。如果您对此还是觉得很模糊，可以想想咱们平时炒菜的过程，一般炒菜时都要放酱油吧，酱油本身就是个高级的东西，它也是被其他的一些农作物制作出来的（比如一般的酱油是用大豆制作的），咱们不也是直接拿来就用吗，有哪位同学因不清楚酱油的制作过程而不敢用酱油啦？炒菜时加酱油和用 C 语言写操作系统是同一个道理，都是以高级的东西为基础来创建新的东西。

如果以上三段内容并没有解开您的疑惑也不要着急，这一切都会在今后的 C 语言编程中理解，由量变到质变，您的问题自然就解决了。

好啦，前面说的太多啦，怕大家着急了，现在赶紧给大家上硬菜，欢迎我们神秘嘉宾——main.c。这是我们第一个 C 语言代码。

代码 5-5　（project/c5/c/kernel/main.c）

```
1 int main(void){
2     while(1);
```

```
3     return 0;
4 }
```

如代码 5-5 所示，它没法再简单啦，简单的程序似乎能帮助咱们更容易地理解所学的知识，哈哈，我说的是似乎，其实，再长的代码，编译后生成的文件结构也是由那几个部分组成，万变不离其宗。这里所说的文件结构是指将来要说的 elf 文件格式，在此不多说，留作伏笔。

正如之前所说，咱们只有用 C 语言的语法结构，这里没有包含标准库，也没有直接的系统调用，以后咱们都得按照这种简洁的方式编程啦。另外，有的同学已经注意到 main.c 所在的目录啦，本来我还想卖个关子的，但它所在的目录出卖了我：在 kernel 目录下。对，如您所想，它就是我们第一个内核文件，我们在 project 目录下建立了个子目录 kernel，今后我们所有与内核相关的模块都要放在此目录下。

您也看到了，这个内核文件什么都没做，通过 while（1）这个死循环一直使用 CPU，目的是停在这里。想当初我就因为忘记加这样的语句而导致不知道 CPU 执行到哪去了，当时排查错误时就晕了，看到执行的指令都不是自己写的，甚至都怀疑是虚拟机的问题，想想好惭愧啊。当然查出来原因之后，自然就是喜极而泣啦。这个简单粗暴可依赖的死循环仅仅是为了演示 elf 文件解析以及加载内核的作用。

生成 C 语言程序的过程是这样的。先将源程序编译成目标文件（由 c 代码变成汇编代码后，再由汇编代码生成二进制的目标文件），再将目标文件链接成二进制可执行文件。平时我们写只有一个文件的小程序时，编译器也是悄悄在背后这样做的，除非加了参数让编译器分成两个动作。由于咱们用的是 C 语言写的程序，想到的编译器自然是大名鼎鼎的 gcc，所以我们用 gcc 编译该程序的参数是：

gcc -c -o kernel/main.o kernel/main.c，也许对其中的参数有的同学不太熟，没关系，在执行 gcc –help 回车后，大家可以看到一些帮助信息，其中：

-c 的作用是编译、汇编到目标代码，不进行链接，也就是直接生成目标文件。

-o 的作用是将输出的文件以指定文件名来存储，有同名文件存在时直接覆盖。

经过上面 gcc 的编译后，我们得到了 main.o 文件，目前为止，它还是个“半成品”。为什么这么说呢？因为它只是个目标文件，也称为待重定位文件，重定位指的是文件里面所用的符号还没有安排地址，这些符号的地址需要将来与其他目标文件“组成”一个可执行文件时再重新定位（编排地址），这里的符号就是指该目标文件中所调用的函数或使用的变量，而这里的“组成”就是指链接。这些符号一般是位于其他文件中，所以在编译时不能确定其地址，需要在所有目标文件都到齐了，将它们链接到一起时再重新定位（编排地址）。由于不知道可执行文件由几个目标文件组成，所以一律在链接阶段对符号重新定位（编排地址）。所以说，哪怕是可执行文件只是由一个文件组成的，其目标文件中的符号也是未编址的，编址工作，即重定位，一律统一在链接阶段完成。

编译成目标文件时，我们可以用 file 命令检查一下 main.o 的状态，如 file kernel/main.o，输出如图 5-26 所示。

```
[work@localhost c]$ file kernel/main.o
kernel/main.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
```

▲图 5-26　目标文件的属性

为了让大家更明显地看出目标文件的可重定位属性，我将 relocatable 用方框给大家圈出来了。

目标文件是可重定位文件，其中的符号都尚未“定位”，也就是符号（变量名，函数名）的地址尚未确定，这一点我们可以用 Linux 的 nm 命令来查看，如图 5-27 所示。

```
[work@localhost c]$ nm kernel/main.o
00000000 T main
```

▲图 5-27　目标文件中符号地址未确定

如图 5-27 所示，由于咱们的 main.c 过于简单，里面只有一个符号，即 main，所以 nm 只列出了它的符号信息。main 函数的地址由于未被指定，所以其值为 00000000。一会儿咱们链接后再对比下大家就更清楚了。

在 Linux 下用于链接的程序是 ld，链接有一个好处，可以指定最终生成的可执行文件的起始虚拟地址。它是用-Ttext 参数来指定的，所以咱们可以执行以下命令完成链接。

ld kernel/main.o -Ttext 0xc0001500 -e main -o kernel/kernel.bin

从左到右说一下参数，-Ttext 指定起始虚拟地址为 0xc0001500，这个地址是设计好的，为什么用这个地址，咱们将来在加载内核时会告诉大家，在此大伙儿先淡定一下。其中–o 的意义也是指定输出的文件名，至于-e，还是要看一下官方帮助。

ld –help 回车后，输出的信息太多，咱们只看下面的-e 参数。

```
-e ADDRESS, --entry ADDRESS Set start address
```

-e 和--entry 一样，字面上的意思是用来指定程序的起始地址。注意，不要被迷惑了，虽然说是指定起始地址，但参数不仅可以是数字形式的地址，而且可以是符号名，这和汇编中的标号也是地址是一样的道理。总之它用来指定程序从哪里开始执行。

为了让大家更清楚-e 的意思，咱们不加-e 参数试试，如图 5-28 所示。

```
[work@localhost c]$ ld kernel/main.o -Ttext 0xc0001500  -o kernel/kernel.bin
ld: warning: cannot find entry symbol _start; defaulting to 00000000c0001500
```

▲图 5-28　ld 链接演示

经过这样的链接操作，ld 报错发出了警告，提示找不到入口符号（entry symbol）_start，默认地址为 00000000c0001500。这个_start 是什么呢？

一个程序总该有个入口地址，这个地址表示的是程序将从哪里开始执行。要知道，并不是程序体中的第一个字节就是程序的起始地址，因为在程序的开头可能有函数声明或数据定义，想想咱们的汇编文件 loader.S，它最前面的部分可不是指令，而是一堆数据，而我们在设计它的时候，知道它的入口地址不在程序开始处，所以在 mbr 中直接跳入了 loader.S 的 loader_start 标号处，跨过了程序开头的数据部分。这还仅仅是由一个 loader.S 生成 loader.bin，并且是我们提前知道入口地址的情况，如果当多个文件拼合成一个可执行文件时，计算机如何知道程序的入口在哪里呢？也就编译后的程序应该从哪句代码开始执行呢？这入口代码可说不准是哪一个了。由于程序内的地址是在链接阶段编排（也就是重定位）的，所以在链接阶段必须要明确入口地址才行，于是链接器规定，默认只把名为_start 的函数作为程序的入口地址，即默认的 entry symbol 是_start，除非另行指定。

大家看到了，代码 5-5 中并没有_start 这个符号，链接器 ld 找不到该起始地址，所以发出了警告。

既然缺少_start 符号，那现在把主函数 main 改成_start 试试，代码如下。

```
1 //int main(void) {
2 int _start(void) {
3      while(1);
4      return 0;
5 }
```

好啦，编译链接一气呵成，整个过程没出任何问题，如图 5-29 所示。

```
[work@localhost c]$ gcc -c -o kernel/main.o  kernel/main.c
[work@localhost c]$ ld kernel/main.o -Ttext 0xc0001500  -o kernel/kernel.bin
[work@localhost c]$ file kernel/kernel.bin
kernel/kernel.bin: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, not stripped
[work@localhost c]$
```

▲图 5-29　把 main 函数名改为_start

在图 5-29 中，我们还用 file 命令查看了最终生成的 kernel.bin 文件，您看，它已经是 executable 啦，即可执行文件。悄悄提示一下，该文件放到虚拟机上运行也是没问题的。

有没有同学想过，这里写一个_start 函数，让其调用 main 函数如何？其实这是可以的，main 函数并不是第一个函数，它实际上也是被别人调用的，不过这是编译器背后的策略啦，好奇心大的同学自己尝试下吧。

虽然把函数名改成_start 可以解决问题，但我们习惯于 main 函数作为主函数，不习惯函数用_start，于是用了-e 来指定起始的函数名为 main，所以代码 5-5 才链接正常。

也许有同学想过，哎？我平时写的程序也没有_start 啊，直接用 gcc 编译后就能运行，没出过问题啊。是啊，确实如您所说，由于我也没深入研究过，但咱们通过比较的方式，让您自己悟出这里面的秘密。

还是以代码 5-5 为例，用 gcc –o /tmp/test.bin kernel/main.c 编译链接，由于未加-c 参数，生成的 test.bin 不再是目标文件，而是可执行文件。然后再用先编译成目标文件再链接成可执行文件的方式，对比这两个文件的区别，如图 5-30 所示。

您看，test.bin 是 gcc 直接生成的可执行文件，它的大小是 4586 字节。而 kernel.bin 是经过手动编译、链接这两个步骤完成的，其文件大小是 1777 字节。这两个文件的体积可是差了几乎 2 倍呢。再看看这两个文件中的符号信息，还是用 nm 命令，如图 5-31 所示。

```
[work@localhost c]$ gcc -o /tmp/test.bin kernel/main.c
[work@localhost c]$ ll /tmp/test.bin
-rwxrwxr-x. 1 work work 4586 10月  12 18:59 /tmp/test.bin
[work@localhost c]$
[work@localhost c]$ gcc -c -o kernel/main.o kernel/main.c
[work@localhost c]$ ld kernel/main.o -Ttext 0xc0001500  -e main -o kernel/kernel.bin
[work@localhost c]$ ll kernel/kernel.bin
-rwxrwxr-x. 1 work work 1777 10月  12 19:00 kernel/kernel.bin
[work@localhost c]$
```

▲图 5-30　自动生成与手动链接的程序对比

```
[work@localhost c]$ nm /tmp/test.bin |wc -l
34
[work@localhost c]$
[work@localhost c]$ nm /tmp/test.bin
        略
        。。。
080482e0 T _start
080495fc b completed.5972
080495f8 W data_start
08049600 b dtor_idx.5974
08048370 t frame_dummy
08048394 T main
[work@localhost c]$
[work@localhost c]$ nm kernel/kernel.bin
c0002505 A __bss_start
c0002505 A _edata
c0002508 A _end
c0001500 T main
[work@localhost c]$
```

▲图 5-31　自动编译链接与手动链接对比

test.bin 中共有 34 个符号（wc –l 命令用来统计输出的行数，一个符号占用一行，故 34 个符号），由于输出太长了，我们只截取了关键的部分，不过您看那些 frame_dummy、data_start 等，这并不是咱们代码中存在的符号，这说明编译器在编译过程中为咱们引用了别的代码，这就是 c 运行库的功劳，目的是在调用 main 函数前做初始化环境等工作。您看，用白色方框圈出来的_start，这就是默认的入口符号，链接器还是用到了它，它不是咱们提供的代码，依然是运行库提供的，这也说明 main 函数不是第一个执行的代码，它一定是被其他代码调用的，main 函数在运行库代码初始化完环境后才被调用。

咱们继续看 kernel.bin 中的符号，一共就 4 行，尽管其中也包含了咱们不认识的符号，但毕竟少得多，我们的程序更短小精干，而且确实没有_start 函数。这里添加了 3 个类型为 A 的符号，这表示它们的值是不变的。T 表示该符号位于代码段中，更多符号的意义请参考 man nm。

其实代码 5-5 要是换成汇编代码的话，就是个 jmp $，其大小不过是 2 字节的机器码 ebfe。除了编译器自动添加的代码外，一般情况下 C 语言编译出来的程序也比汇编语言生成的程序体积大。可见，人们常说的汇编语言比 C 语言快，并不是汇编语言本身有多快（它也要变成机器指令后才能上 CPU 运行），而是汇编语言对应的机器指令是一对一的，简单，直接，可依赖，而 C 语言生成的机器指令是一对多的，复杂，间接，略冗余。

好啦，关于内核的部分咱们就此先打住，其实说这话我有点不好意思，您也看到啦，内核代码中就一个死循环而已，我们的内核还没有开始，请无视我吧。咱们的内核虽然离真正的内核差得十万八千里，但它目的是两个：一是为了演示加载内核，二是为了演示 elf 格式的文件解析。后面我们将结合此简单至极的 c 程序来学习有关 elf 方面的知识。

5.3.2　二进制程序的运行方法

操作系统并不是在功能上给予用户的支持，这种支持体现在机制上。也就是说，单纯的操作系统，用户拿它什么都做不了，用户需要的是某种功能。而操作系统仅仅是个提供支持的平台。

虽然我们是模仿 Linux 来写一个黑屏白字的系统，但如果没有 Windows 的话，估计当今这个世界将会失去 70%以上的光芒。由于有了操作系统的支持，我们可以安装一些软件，也就是应用程序，比如安装了 QQ 或其他一些的即时通信工具，这样我们就能够同其他人聊天。

所以，操作系统并不能直接帮大家做什么，但大家想做什么的时候，操作系统能提供最大限度的支持。

任何程序都需要被载入到内存后才能运行，这是 CPU 等其他硬件的运行机制决定的，我们若在该硬件系统上运行程序，不得不遵守这样那样的约束。应用程序是独立于操作系统的，它不会像操作系统那样，含着金钥匙，一出生就直接在内存中。它们通常位于磁盘等外存设备中，在使用时，需要从外存中将其调入到内存后才行。

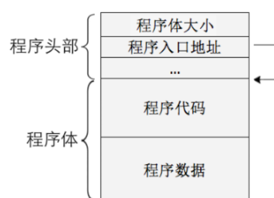
如何去加载用户程序呢？

操作系统是程序，是软件，用户程序也是软件，用一个程序去调用另一个程序一点难度都没有，最最简单的办法，就是用 jmp 或 call 指令。我们的 BIOS 就是这样调用 mbr 的，我们的 mbr 就是这样调用 loader 的。但大家还记得不，BIOS 调用 mbr，mbr 的地址是 0x7c00，mbr 调用 loader，loader 的地址是 0x900。这两个地址是固定的，也就是说，我们目前的方法是很不灵活的，调用方需要提前和被调用方约定调用地址。

有没有一种灵活的方法让程序的加载地址不那么固定呢？

显然是有的，由于每个程序是单独存在的，所以程序的入口地址信息需要与程序绑定，最简单的办法就是在程序文件中专门腾出个空间来写入这些程序的入口地址，主调程序在该程序文件的相应空间中将该程序的入口信息读出来，将其加载到相应的入口地址，跳转过去就行了。当然不仅仅只写入程序入口地址，能写的东西很多，比如为了给程序分配内存，至少还得需要知道程序的尺寸大小。但在哪里写入程序的入口地址呢？这便是文件头的由来，在程序文件的开头部分记载这类信息，而程序文件中除文件头外其余的部分则是之前的程序体。这样一来，原先的纯二进制可执行文件加上新的文件头，就形成了一种文件格式。不仅文件是这样，很多其他传输协议也是采用文件头 header+文件体 body 的形式，如邮件传输协议和 http 传输协议。在现实生活中也有这样的例子，比如咱们坐火车的时候，按理说，只要火车停在能让咱们看到的地方，咱们就能直接上火车了。但现实中不可能让所有火车摆在咱们面前，所以我们在乘坐火车时，都是进站后先要查看大屏幕上的列车时刻表，从中找到在哪个候车室等候上车。其中，列车时刻表就相当于文件头，我们从中找到上火车的入口，而火车则相当于文件体。

在程序中，程序头（也就是文件头）用来描述程序的布局等信息，它属于信息的信息，也就是元数据。包含程序头的程序文件示意如图 5-32 所示。



▲图 5-32　包含程序头的程序文件

由于程序文件中包含了程序头，好处是程序的入口地址等信息不需要写死，调用方中的调用代码可以变得通用，根据实际情况加载便可。但不好的地方是这些元信息不是代码，故不应该将其放在 CPU 上“执行”，所以程序就不再是纯粹的二进制可执行文件了，不像之前咱们用 nasm 默认编译的可执行文件（里面全是程序本身的指令和数据）那样纯粹。所以，将这种具有程序头格式的程序文件从外存读入到内存后，从该程序文件的程序头中读出入口地址，需要直接跳进入口地址执行，跨过程序头才行。

程序头可以自定义，只要我们按照自己定义的格式去解析就行。也许我光这么一说，很多同学还是不能彻底明白如何自定义文件头，因为大多数同学都是用高级语言来写程序的，即使用了偏底层的 C 语言，不同平台的 c 编译器也会根据系统平台自动添加文件头，不给咱们亲手体验自定义程序头的机会。汇编语言非常灵活，所以用它来构建任意文件格式是非常方便的。书看到这里，我估计您已经发现我是个非常体贴的人，哈哈，所以给大家呈上以下代码来演示自定义文件头，请见汇编程序代码 5-6。

代码 5-6　header.S（测试代码，无实例）

```
[work@localhost book]$ cat -n header.S
     1 header:
     2        program_length      dd    program_end-program_start
     3        start_addr      dd    program_start
     4 ;;;;;;;     以上是文件头，以下是文件体     ;;;;;;;
     5 body:
     6 program_start:
     7     mov ax, 0x1234
     8     jmp $
     9 program_end:
    10
[work@localhost book]$
```

以上测试代码 header.S，经编译后生成的文件是 header.bin，编译命令是 nams –o header.bin header.S，所以 header.bin 依然是纯二进制文件。但此纯二进制文件的程序入口并不是文件开头，这和咱们的 loader.bin 很像，mbr 是跳到 loader.bin 的中间某部分去执行。假设 header.S 是被调用的程序，调用方知道 header.S 的前 8 字节是程序头，在这 8 字节中，前 4 字节用来标明程序尺寸大小，后 4 字节用来指明程序的入口地址，也就是该程序的第一条指令地址。从 8 字节后到文件结束为文件体。在调用方程序已经了解此文件格式后，它可以这样做。

（1）将 header.bin 前 8 字节的内容读到内存，前 4 字节是程序体长度，后 4 字节是程序的入口地址。

（2）将 header.bin 开头偏移 8 字节的地方作为起始，将 header.bin 文件尾，即开头偏移（8+程序体长度）个字节的地方作为终止。

（3）将起始至终止之间的程序体复制到入口地址。

（4）转到入口地址处执行。

您看，被调用方 header.bin 被设计成这样的文件格式，调用方就可以自由处理啦。为了验证文件格式，大家看下 header.bin 生成的二进制文件真实内容，前 8 字节用了两个下划线分别区分了两个程序头属性，即 program_length 是 0x00000005，start_addr 是 0x00000008。顺序是反着的，小端字节序。B83412 是 mov ax，0x1234 指令，EBFE 对应的是 jmp $，如图 5-33 所示。

```
[work@localhost book]$ sh ~/tool/xxd.sh header.bin 0 300
0000000: 05 00 00 00 08 00 00 00 B8 34 12 EB FE         .........4...
[work@localhost book]$
```

▲图 5-33　自定义程序头的可执行文件

当然这仅仅是用来演示的，目的是起到抛砖引玉的作用，说了这些是为了引出后面咱们要介绍的文件格式。

自己设计的文件头自己当然认识，但这毕竟不通用，我们需要选择一种流行的文件格式，咱们是在 Linux 下用 C 语言编程，其编译器 gcc 生成的是 elf 文件格式，咱们在下一节展开 elf 可执行文件格式的内容。

5.3.3　elf 格式的二进制文件

一开始我就在想，如何将此 elf 文件格式讲清楚，回想起我们在上大学的时候，操作系统课程的老师一般都是这样介绍 ELF 的：Window 下的可执行文件格式是 PE（如果您想说的是 EXE，不要搞混了，EXE 是扩展名，属于文件名的一部分，只是名字的后缀，它并不是真正的格式），PE 即 Portable Executable，Linux 下可执行文件格式是 ELF。一般只是通过这样简单的对比来认识 elf 格式的，这充其量算是个简介，根本谈不上详实。我想，那时的我们很少有人能从课堂上明白 ELF 文件格式的本质。本文不再重蹈覆辙，咱们不求深入剖析，也要采取“浅析”的方式来阐述。

ELF 指的是 Executable and Linkable Format，可执行链接格式。最初是由 UNIX 系统实验室（USL）作为应用程序二进制接口（ABI）而开发和发行的。工具接口标准委员会（TIS）选择了它作为 IA32 体系结构上不同操作系统之间的可移植二进制文件格式，于是它就发展成为了事实上的二进制文件格式标准。

先跟大家交待下，在 ELF 规范中，把符合 ELF 格式协议的文件统称为“目标文件”或 ELF 文件，这与我们平时所说的目标文件是不同的。

在大家平时的习惯中，咱们把编译后，但未经链接的文件称为目标文件，也称为待重定位文件（relocatable file），比如在 Linux 下用 gcc –c 参数生成的.o 文件。而平时我们所说的 ELF 文件也是指经过编译链接后的二进制可执行文件，该文件能够直接运行。

为了避免混淆，咱们采用与 ELF 规范相同的命名方式，本节中所说的目标文件即指各种类型符合 ELF 规范的文件，如二进制可执行文件和 Linux 下.o 结尾的目标文件和.so 结尾的动态库文件。而待重定位文件，可以理解成咱们惯常所说的目标文件（如 Linux 下的.o 文件）。ELF 目标文件归纳见表 5-7。

表 5-7　　elf 眼中的目标文件

ELF 目标文件类型	描　　述
待重定位文件（relocatable file）	待重定位文件就是常说的目标文件，属于源文件编译后但未完成链接的半成品，它被用于与其他目标文件合并链接，以构建出二进制可执行文件或动态链接库。为什么称其为“待重定位”文件呢？原因是在该目标文件中，如果引用了其他外部文件（其他目标文件或库文件）中定义的符号（变量或者函数统称为符号），在编译阶段只能先标识出一个符号名，该符号具体的地址还不能确定，因为不知道该符号是在哪个外部文件中，而该外部文件需要被重定位后才能确定文件内的符号地址，这些重定位的工作是需要在连接的过程中完成的
共享目标文件（shared object file）	这就是我们常说的动态链接库。在可执行文件被加载的过程中被动态链接，成为程序代码的一部分
可执行文件（executable file ）	经过编译链接后的、可以直接运行的程序文件

为什么先给大家介绍这些，在后面大家就知道了，elf 各种数据结构中牵扯到各种“类型”，已经了解的同学就忽略我吧，这是给那些没接触过此方面的同学准备的，先让大家有个感性认识。

本来我想和大家提前约定一下本文所用到的术语风格，其实我想一律用英文单词，但担心大家如果

某天翻看到中间某页，无法将英文解释对号入座，所以我尽量适时地穿插中英文同时标注。

好，为了描述清楚文件格式的本质，咱们先从最基本的“段”说起。

程序中最重要的部分就是段（segment）和节（section），它们是真正的程序体，是真真切切的程序资源，所以下面的说明咱们以它们为例。程序中有很多段，如代码段和数据段等，同样也有很多节，段是由节来组成的，多个节经过链接之后就被合并成一个段了，之前咱们有通过实例解释过 segment 和 section 之间的关系。

段和节的信息也是用 header 来描述的，程序头是 program header，节头是 section header。

程序中段的大小和数量是不固定的，节的大小和数量也不固定，因此需要为它们专门找个数据结构来描述它们，这个描述结构就是程序头表（program header table）和节头表（section header table）。既然程序头表和节头表都称为表，这说明里面存储的是多个程序头 program header 和多个节头 section header 的信息，故这两个表相当于数组，数组元素分别是程序头 program header 和节头 section header。再次强调，这两个表是用来将汇总程序头和节头的表，表中元素是头信息。也就是说程序头表（program header table）中的元素全是程序头（program header），而节头表（section header table）中的元素全是节头（section header）。虽然上面是将两个表一块说明的，但表中的元素全是单一的，不会在程序头表中存在节头信息。

在表中，每个成员（数组元素）都统称为条目，即 entry，一个条目代表一个段或一个节的头描述信息。对于程序头表，它本质上就是用来描述段（segment）的，所以您也可以称它为段头表。从名字上就能够看出，段等同于程序，所以将描述段信息的表说成 program header table，可见“段”才是程序本身的组成部分。

由于程序中段和节的数量不固定，程序头表和节头表的大小自然也就不固定了，而且各表在程序文件中的存储顺序自然也要有个先后，故这两个表在文件中的位置也不会固定。因此，必须要在一个固定的位置，用一个固定大小的数据结构来描述程序头表和节头表的大小及位置信息，这个数据结构便是 ELF header，它位于文件最开始的部分，并具有固定大小，一会儿咱们看 elf header 的数据结构就知道了。

ELF header 是个用来描述各种“头”的“头”，程序头表和节头表中的元素也是程序头和节头，可见，elf 文件格式的核心思想就是头中嵌头，是种层次化结构的格式。

有了上面的宏观介绍，下面就好理解多了。

ELF 文件格式依然分为文件头和文件体两部分，只是该文件头相对稍显复杂，类似层次化结构，先用个 ELF header 从“全局上”给出程序文件的组织结构，概要出程序中其他头表的位置大小等信息，如程序头表的大小及位置、节头表的大小及位置。然后，各个段和节的位置、大小等信息再分别从“具体的”程序头表和节头表中予以说明。

ELF 格式的作用体现在两方面，一是链接阶段，另一方面是运行阶段，故它们在文件中组织布局咱们也从这两方面展示，如图 5-34 所示。

链接视图	运行视图
ELF header（elf 头）	ELF header（elf 头）
Program header table（程序头表） 可选	Program header table（程序头表）
Section 1（节 1）	Segment 1（段 1）
...	
...	Segment 2（段 2）
Section n（节 n）	
...	...
Section header table（节头表）	Section header table（节头表） 可选
...	...
待重定位文件体	可执行文件体

▲图 5-34　elf 文件格式布局

如图 5-34 所示，无论是在待重定位文件，还是可执行文件中，文件最开头的部分必须是 elf header，这就是前面咱们所说的固定的位置。在 ELF header 之后紧挨着的是程序头表，这对于可执行文件是必须存在的，而对于待重定位文件是可选的。其他成员的位置要取决于各头表中的说明。坦白说，刚接触 elf 之初并不容易理解它，所以，之后咱们会以一个实际例子来细说，这里咱们先有个笼统的认识。

咱们马上要步入正题啦，在此之前必须要提前跟大伙儿交待一下，以下咱们书中有关 elf 的任何定义，包括变量、常量及取值范围，都可以在 Linux 系统的/usr/include/elf.h 中找到，这里面的定义才是最全最权威的。为了方便大家学习，本书从 elf.h 中搬了必要的部分，有的并不全面，只是起到帮助大家阅读本书的作用。

一切就绪，咱们开始正式介绍 elf 结构。

一些重要的数据结构中用到了自定义的数据类型，所以先给大家介绍一下它们，免得造成学习的困扰，见表 5-8 所列。

表 5-8　　elf header 中的数据类型

数据类型名称	字节大小	对　　齐	意　　义
Elf32_Half	2	2	无符号中等大小的整数
Elf32_Word	4	4	无符号大整数
Elf32_Addr	4	4	无符号程序运行地址
Elf32_Off	4	4	无符号的文件偏移量

好啦，现在咱们从上至下，依次揭开各层 header 的庐山真面目。咱们这里先介绍 ELF header 的结构。

C 语言中的结构体能够很直观地表示物理内存结构，用结构体的形式展现一个数据结构是最合适不过的啦，所以咱们结合图 5-35，依次介绍下各结构体成员的意义。

```
struct Elf32_Ehdr {
  unsigned char  e_ident[16];
  Elf32_Half     e_type;
  Elf32_Half     e_machine;
  Elf32_Word     e_version;
  Elf32_Addr     e_entry;
  Elf32_Off      e_phoff;
  Elf32_Off      e_shoff;
  Elf32_Word     e_flags;
  Elf32_Half     e_ehsize;
  Elf32_Half     e_phentsize;
  Elf32_Half     e_phnum;
  Elf32_Half     e_shentsize;
  Elf32_Half     e_shnum;
  Elf32_Half     e_shstrndx;
};
```

▲图 5-35　elf header 结构

e_ident[16]是 16 字节大小的数组，用来表示 elf 字符等信息，开头的 4 个字节是固定不变的，是 elf 文件的魔数，它们分别是 0x7f，以及字符串 ELF 的 asc 码：0x45, 0x4c, 0x46。对于此数组说明见表 5-9。

表 5-9　　e_ident 数组功能简介

e_ident 数组成员	意　　义
e_ident[0] = 0x7f	这 4 位是固定的 ELF 文件的魔数，如果它们的值如左列 4 行所示，表明这就是一个 ELF 文件
e_ident[1] = 'E'	
e_ident[2] = 'L'	
e_ident[3] = 'F'	
e_ident[4]	用来标识 elf 文件的类型 值为 0 表示该文件是不可识别类型 值为 1 表示该文件是 32 位 elf 格式的文件 值为 2 表示该文件是 64 位 elf 格式的文件
e_ident[5]	用来指定编码格式，其实就是指定大端字节序还是小端字节序 值为 0 表示非法编码格式 值为 1 表示小端字节序，即 LSB（最低有效字节） 值为 2 表示大端字节序，即 MSB（最高有效字节）
e_ident[6]	ELF 头的版本信息，默认为 1 值为 0 表示非法版本 值为 1 表示当前版本
e_ident[7～15]	暂且不用，保留，均初始化为 0

在这里插播个小插曲，有关 e_ident[5]大小端字节序，和大家分享一个技巧，用 file 命令就能够查看到 elf 格式的可执行程序是 LSB，还是 MSB。

例如 file /bin/ls 回车后，输出信息为：

/bin/ls: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.18, stripped。

我想，您已经注意到了在 32-bit 后面的 LSB。好啦，咱们继续说其他属性。

e_type 占用 2 字节，是用来指定 elf 目标文件的类型，可能的取值见表 5-10。

表 5-10　　elf 目标文件类型

elf 目标文件类型	取　　值	意　　义
ET_NONE	0	未知目标文件格式，忽略
ET_REL	1	可重定位文件

续表

elf 目标文件类型	取　　值	意　　义
ET_EXEC	2	可执行文件
ET_DYN	3	动态共享目标文件
ET_CORE	4	core 文件，即程序崩溃时其内存映像的转储格式，俗称出 core 了
ET_LOPROC	0xff00	特定处理器文件的扩展下边界
ET_HIPROC	0xffff	特定处理器文件的扩展上边界

表 5-10 列出了许多类型文件，前 5 种类型文件看上去稍显“亲民”还能接受。最后两种，ET_LOPROC 和 ET_HIPROC 这两个类型的取值跨度好大，显得似乎有些怪异，其实把它们搞得如此怪异，是为了突显它们的“与众不同”，它们是与硬件相关的参数，在它们之间的取值用来标识与处理器相关的文件格式。不过呢，虽然有这么多类型，但咱们只需要关注取值为 2 的 ET_EXEC 类型，它的意义为程序可执行，就是咱们平时编译链接好的可执行程序的类型。

e_machine 占用 2 字节，用来描述 elf 目标文件的体系结构类型，也就是说该文件要在哪种硬件平台（哪种机器）上才能运行。可能的取值见表 5-11。

表 5-11　　elf 目标文件所属的体系结构类型

体系结构类型	取　　值	意　　义
EM_NONE	0	未指定
EM_M32	1	AT&T WE 32100
EM_SPARC	2	SPARC
EM_386	3	Intel 80386
EM_68K	4	Motorola 68000
EM_88K	5	Motorola 88000
EM_860	7	Intel 80860
EM_MIPS	8	MIPS RS3000

从表 5-11 中列出的 7 种体系结构可以看出，elf 实现了机器平台无关的良好可移植性。

e_version 占用 4 字节，用来表示版本信息。

e_entry 占用 4 字节，用来指明操作系统运行该程序时，将控制权转交到的虚拟地址。

e_phoff 占用 4 字节，用来指明程序头表（program header table）在文件内的字节偏移量。如果没有程序头表，该值为 0。

e_shoff 占用 4 字节，用来指明节头表（section header table）在文件内的字节偏移量。若没有节头表，该值为 0。

e_flags 占用 4 字节，用来指明与处理器相关的标志，本书用不到那么多的内容，具体取值范围，有兴趣的同学还是要参考/usr/include/elf.h。

e_ehsize 占用 2 字节，用来指明 elf header 的字节大小。

e_phentsize 占用 2 字节，用来指明程序头表（program header table）中每个条目（entry）的字节大小，即每个用来描述段信息的数据结构的字节大小，该结构是后面要介绍的 struct Elf32_Phdr。

e_phnum 占用 2 字节，用来指明程序头表中条目的数量。实际上就是段的个数。

e_shentsize 占用 2 字节，用来指明节头表（section header table）中每个条目（entry）的字节大小，即每个用来描述节信息的数据结构的字节大小。

e_shnum 占用 2 字节，用来指明节头表中条目的数量。实际上就是节的个数。

e_shstrndx 占用 2 字节，用来指明 string name table 在节头表中的索引 index。

接下来再给大家介绍下程序头表中的条目的数据结构，这是用来描述各个段的信息用的，其结构名

为 struct Elf32_Phdr。怕有同学搞混了，在此容我啰嗦一下，此段是指程序中的某个数据或代码的区域段落，例如数据段或代码段，并不是内存中的段，到现在为止我们都在讨论位于磁盘上的程序文件呢。struct Elf32_Phdr 结构的功能类似 GDT 中段描述符的作用，段描述符用来描述物理内存中的一个内存段，而 struct Elf32_Phdr 是用来描述位于磁盘上的程序中的一个段，它被加载到内存后才属于 GDT 中段描述符所指向的内存段的子集。好啦，话说得有点多啦，其结构如图 5-36 所示。

```
struct Elf32_Phdr {
   Elf32_Word    p_type;
   Elf32_Off     p_offset;
   Elf32_Addr    p_vaddr;
   Elf32_Addr    p_paddr;
   Elf32_Word    p_filesz;
   Elf32_Word    p_memsz;
   Elf32_Word    p_flags;
   Elf32_Word    p_align;
};
```

▲图 5-36　program header 结构

还是按照惯例，咱们先把属性都介绍一下。

p_type 占用 4 字节，用来指明程序中该段的类型。p_type 类型说明见表 5-12。

表 5-12　程序中的段类型

类　型	取　值	说　明
PT_NULL	0	忽略
PT_LOAD	1	可加载程序段
PT_DYNAMIC	2	动态链接信息
PT_INTERP	3	动态加载器名称
PT_NOTE	4	一些辅助的附加信息
PT_SHLIB	5	保留
PT_PHDR	6	程序头表
PT_LOPROC	0x70000000	此范围内的类型预留给处理器专用
PT_HIPROC	0x7fffffff	

p_offset 占用 4 字节，用来指明本段在文件内的起始偏移字节。

p_vaddr 占用 4 字节，用来指明本段在内存中的起始虚拟地址。

p_paddr 占用 4 字节，仅用于与物理地址相关的系统中，因为 System V 忽略用户程序中所有的物理地址，所以此项暂且保留，未设定。

p_filesz 占用 4 字节，用来指明本段在文件中的大小。

p_memsz 占用 4 字节，用来指明本段在内存中的大小。

p_flags 占用 4 字节，用来指明与本段相关的标志，此标志取值范围见表 5-13。

表 5-13　p_flags 取值范围

类　型	取　值	说　明
PF_X	1	本段具有可执行权限
PF_W	2	本段具有可写权限
PF_R	4	本段具有可读权限
PF_MASKOS	0x0ff00000	本段与操作系统相关
PF_MASKPROC	0xf0000000	本段与处理器相关

p_align 占用 4 字节，用来指明本段在文件和内存中的对齐方式。如果值为 0 或 1，则表示不对齐。否则 p_align 应该是 2 的幂次数。

链接后，程序运行的代码、数据等资源都是在段中，所以，在 elf header 和 program header 介绍完后，基本上就已经把与“段”相关的内容说完啦。咱们还是本着“够用就行”的原则学习，有关“section 节”或其他方面的内容咱们这里就不需要关注太多了，有兴趣的同学可以自行深入研究。

以上的说明似乎显得过于抽象，我能想像，也许有的同学开始有点不耐烦了，不过不要着急，之前咱们就说过啦，要拿个实际的例子来解释这些看似复杂的结构，我相信通过实例来解释以上内容，大家一定会茅塞顿开。更多精彩请看下一节。

5.3.4　elf 文件实例分析

在上一节中，我们讲述了 elf 格式的部分理论知识，为什么是部分呢？因为我们本着“够用”的原则，只把我们需要了解的部分说完啦。不过，我相信大部分同学仅仅凭上一节中的理论知识还是领悟不到 elf 本质，咱们在本节开始分析前面咱们写过的“内核”（代码 5-5），让大家看清 elf 文件的每一个字节。

为了让大家看清楚 elf 文件内部，咱们要用之前的 xxd 命令，为了方便使用，如很久以前所述，已经将其封装成了 xxd.sh 脚本，参数 1 是待查看的文件名，参数 2 是文件内的起始字节，参数 3 是查看的连续字节数。脚本是逐字节输出文件的内容。脚本内容很简单，就是 xxd 命令而已：xxd -u -a -g 1 -s $2 -l $3 $1，您也看到了，参数比较多，弄成脚本完全是为了避免每次复杂的参数键入。为了让大家方便使用，我已经将其放到了 tool 目录下，脚本中有参数说明，这里不再列出。下面是用此脚本处理 kernel.bin 的输出，如图 5-37 所示。

```
[work@localhost c]$ sh ~/tool/xxd.sh kernel/kernel.bin 0 300
0000000: 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00  .ELF............
0000010: 02 00 03 00 01 00 00 00 00 15 00 C0 34 00 00 00  ............4...
0000020: 5C 05 00 00 00 00 00 00 34 00 20 00 02 00 28 00  \.......4. ...(.
0000030: 06 00 03 00 01 00 00 00 00 00 00 00 00 10 00 C0  ................
0000040: 00 10 00 C0 05 05 00 00 05 05 00 00 05 00 00 00  ................
0000050: 00 10 00 00 51 E5 74 64 00 00 00 00 00 00 00 00  ....Q.td........
0000060: 00 00 00 00 00 00 00 00 00 00 00 00 06 00 00 00  ................
0000070: 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
0000120: 00 00 00 00 00 00 00 00 00 00 00 00              ............
[work@localhost c]$
```

▲图 5-37　elf 格式剖析

之前我们就用过多次 xxd 命令啦，对于输出想必大家一定很熟悉啦。脚本的输出大概分了三部分，最左边的一列是十六进制的地址，或者称为偏移量最为恰当。中间这一大块矩阵似的十六进制数字是文件中的内容，每两位十六进制数字为一字节，每行共 16 个字节。最右边那一列，含有点点的、偶尔伴有可读字符的部分是字符显示区，这部分将内容按照字符编码显示，当然，前提肯定得是可打印字符，控制字符肯定不行，所以只要不是可显示的字符便显示为'. '。

为了方便大家查看 elf 文件中各部分属性，我在各属性下面用下划线予以区分。其中，细下划线属于 elf header 的范围，粗下划线属于 program header table 程序头表的范围。在各范围之中的各属性，又以明显的下划线分隔，相信大家一定能一目了然。

咱们按照从上到下的顺序，先从细下划线的 elf header 部分说起。

第一行是 e_ident 数组，前 4 字节是固定的 elf 魔数，正如您看到的，它们是 0x7f 和字符 ELF 的 ASCII：0x45、0x4c、0x46。所以您在显示区看到了 ELF 的三个字符。紧跟其后的三个 1 分别是 e_ident[4]、e_ident[5]、e_ident[6]三个成员，代表的意义是 32 位 elf 文件、小端字节序、当前版本。后面的 9 个 00 是 e_ident[7]～e_ident[15]，这些确实都已经初始化为 0。

现在看第二行。

第 1 个下画线处的内容是 02 00，由于是小端字节序，所以其值为 0x0002。以下为方便陈述，只说该字节序所表示的数值。这个是 e_type 属性，它占 2 字节，值为 2 表示类型为 ET_EXEC，即可执行文件（有兴趣的同学可以自行查看 Linux 下的.o 目标文件，其 e_type 类型是值为 1 的 ET_REL，即待重定位类型）。

本行的第 2 个下画线处的内容是 0x0003，占 2 字节。该位置是 e_machine 属性，即 EM_386，表示该 elf 文件是运行在 Intel 80386 平台。

第 3 个下画线处的内容是 0x00000001，占 4 字节。该位置是 e_version 属性，即版本信息。

第 4 个下画线处的内容是 0xc0001500，占 4 字节，该位置是 e_entry 属性，即程序的虚拟入口地址。

第 5 个下画线处的内容是 0x00000034，占 4 字节，该位置是 e_phoff 属性，表示 program header table 程序头表在文件中的偏移量，这里的偏移量是 0x34。

现在看第三行。

第 1 个下画线处的内容是 0x0000055c，占 4 字节，该位置是 e_shoff，表示 section header table 节头表

在文件内的偏移量，这里的值为0x55c，表示在本文件偏移0x55c字节处为节头表。之前说过啦，若没有节头表，此处便为0。

第2个下画线处的内容是0x00000000，占4字节，该位置是e_flags属性。

第3个下画线处的内容是0x0034，占2字节，该位置是e_ehsize属性，表示elf header大小是0x34字节。这和前面e_phoff属性值大小一致，可见，程序头表紧跟着elf header之后。

第4个下画线处的内容是0x0020，占2字节，该位置是e_phentsize属性，即program header的结构：struct Elf32_Phdr的字节大小，值为0x20字节。

第5个下画线处的内容是0x0002，占2字节，该位置是e_phnum属性，即程序头表中段的个数，这里为2个段。

第6个下画线处的内容是0x0028，占2字节，该位置是e_shentsize属性，即节头表中各个节的大小。

现在看第四行。

第1个下画线处的内容是0x0006，占2字节，该位置是e_shnum属性，即节头表中节的个数，这里表示有6个节。

第2个下画线处的内容是0x0003，占2字节，该位置是e_shstrndx属性，即string name table在节头表中的索引为3。

现在开始分析粗下画线范围的程序头表部分。

从第4行到第8行是程序头表的范围，前面说过啦，程序头表中共有2个段，每个段大小是0x20字节。这有两个粗下画线，每个占0x20字节。大家注意图中，在两个粗下画线间有个小竖线，这是用来区分两个段的。竖线左右两边各是一个段。

下面咱们按照struct Elf32_Phdr结构来分析，该结构中每个属性都占4字节，不再赘述。现在还是继续说图5-37的第4行。

第1个粗下画线值为0x00000001，该位置是p_type属性，值为1，即表示PT_LOAD类型，可加载程序段，由于kernel.bin已经是链接后的可执行程序啦，所以，这PT_LOAD类型符合我们的认知。

第2个粗下画线值为0x00000000，该位置是p_offset属性，表示本段在文件内的偏移量。这个偏移量为0，似乎很奇怪，这表示该段的起始是从文件头开始也算起啦，文件开头的部分不是elf header吗？不是代码啊，这是要闹哪样？好吧，到底是什么情况，一会儿咱们细说。

第3个粗下画线值为0xc0001000，该位置是p_vaddr属性，表示本段被加载到内存后的起始虚拟地址。看到这里，似乎觉得上面的p_offset为0有那么一点合理啦，结合elf header中的e_entry的值为0xc0101500，不知您想到了点什么？咱们先把剩下的说完。

第4个粗下画线值为0xc0001000，该位置是p_paddr属性，它通常和p_vaddr值一致，但该属性是保留项，咱们不用关注。

第5个粗下画线值为0x00000505，该位置是p_filesz属性，表示本段在文件中的字节大小。

第6个粗下画线值也应该是0x00000505，该位置是p_memsz属性，表示本段在内存中的大小，因为段无论在哪里，逻辑大小是不变的，故该值等于p_filesz。

第7个粗下画线值为0x00000005，该位置是p_flags属性，表示与本段相关的标志。5=4+1=PF_R+PF_X，在此表示可读，可执行，根据此属性，我们推测此段为代码段。

第8个粗下画线值为0x00001000，该位置是p_align属性，表示本段对齐的方式。

第一个段咱们说完了，第二个段这里就不解释啦，留着大家自己练手吧。咱们现在解决第一个段的p_offset为0的疑惑。

按理说，或者按咱们想像来说，p_offset的值不应该为0，其值至少要跨过elf header和程序头表，这还是在没有节（section）的情况下。虽然此程序kernel.bin在实际加载运行时一点问题都没有（我试过啦，哈哈），但既然是来学习的，有疑惑咱们必须得搞清楚。现在咱们根据实际情况试着推测一下，已知条件：

- 程序的入口地址e_entry的值为0xc0101500。
- 程序的第一段在内存中的虚拟地址p_vaddr的值为0xc0001000。

- 程序的第一段的大小 p_filesz 是 0x00000505。

由这三个已经知条件，至少能看出两件事：

- 由前两个已知条件，能看出程序的入口虚拟地址（p_entry）0xc0101500 跨过了第一个段的开头部分（p_vaddr）0xc0001000，还超过了 0x500 字节，这总让咱们放点心了，至少不是把 elf_header 也当作代码执行啦。
- 第三个已知条件说该段长度是 0x505，这表示该段的最后一字节是 0xc0001000+0x505=0xc0001505。程序的起始地址是 0xc0001500，该段又是代码段（从该段的段标志 p_flags 值为 5 看出：可读可执行）这说明该段的实际代码长度是 0xc0001505-0xc0101500=5 字节。

排除疑惑最简单的方法，就是验证下地址 0xc0101500 处的代码是不是正确的。代码 5-5 中的 while（1）；这句话是个死循环，想像一下，如果这个死循环对应到汇编代码，八成是变成“jmp 标号”类似这样的语句，而这个标号在编译过程中会被编译器替换为具体的地址。我们先看一下是不是这样，怎么看呢？方法很多，这里咱们用 gcc 提供的方法。咱们之前说过啦，c 代码会先被转换成汇编代码，这个中间过程如果不干涉编译器的话咱们是看不到的，转瞬即逝，至少 gcc 编译会在临时目录中建立临时文件，使用完成后会清理得干干净净。如果大家感兴趣，在 gcc 编译时可以加个参数-v（verbose），这样就会输出冗余的内容，从而展示出编译过程的更多细节。话说多啦，咱们要想查看转换后的汇编代码，要加个-S 参数，这个参数告诉 gcc：转换成汇编后就停止，不再进行编译链接。大家可以通过 gcc --help|grep '\-S'来查看。

咱们如此操作一下，过程如图 5-38 所示。

```
[work@localhost kernel]$ cat -n main.c
     1  int main(void) {
     2     while(1);
     3     return 0;
     4  }
[work@localhost kernel]$
[work@localhost kernel]$ gcc -S -o /tmp/main.S main.c
[work@localhost kernel]$ cat -n /tmp/main.S
     1          .file   "main.c"
     2          .text
     3  .globl _start
     4          .type   _start, @function
     5  _start:
     6          pushl   %ebp
     7          movl    %esp, %ebp
     8  .L2:
     9          jmp     .L2
    10          .size   _start, .-_start
    11          .ident  "GCC: (GNU) 4.4.6 20120305 (Red Hat 4.4.6-4)"
    12          .section        .note.GNU-stack,"",@progbits
[work@localhost kernel]$
```

▲图 5-38　main.s 生成的汇编代码 main.S

如图 5-38 所示，main.c 最终生成的汇编代码 main.S 有 12 行，其中实际的指令部分只有三行，我已经用方框框出来了，里面的汇编代码风格是 AT&T。

.text 表示下面开始定义代码，所以从第 2 行的.text 看出，main.c 确实被汇编成了代码段，由 gcc 编译在第 3 行将_start 导出为全局符号，并在第 4 行声明_start 是个函数。方框中的第 5～7 行是一些准备工作，属于堆栈框架的例行公事，之前咱们也提到过它，之前也说过以后会讲，不过不是现在，还没到时候。相信我，以后咱们会结合实例讲述它的，目前咱们不管它啦。重点是第 7～8 行的标号和代码，jmp .L2。它们就是对应于 C 语言中的 while(1)。

答案也差不多能揭晓了，现在只要看看实际可执行文件中的机器码就行了，我们查看一下在 kernel.bin 中起始虚拟地址处的内容。

起始虚拟地址只是个对应于内存中的地址，程序在内存中才用得着它，现在我们通过虚拟地址来计算它在文件内的位置，也就是需要将其转换成在文件中的偏移量。程序的入口虚拟地址 p_entry 是 0xc0101500，第一个段的起始虚拟地址 p_vaddr 为 0xc0001000，并且第一个段在文件内的偏移量为 0，故，起始虚拟地址 e_entry 对应在文件中的偏移量为 0xc0101500-0xc0001000+ 0=0x500。将其换算成十进制为 1280。还是小心起见，这个偏移量肯定不能超过文件大小，该文件大小为 1777 字节，验证通过。这下可以用 xxd.sh 脚本检查 kernel.bin 偏移为 1280 处的 5 个字节啦。过程如图 5-39 所示。

```
[work@localhost kernel]$ ll kernel.bin
-rwxrwxr-x. 1 work work 1777 10月  13 21:36 kernel.bin
[work@localhost kernel]$ sh ~/my_workspace/tool/calculator.sh 0x500 d
1280
[work@localhost kernel]$ sh ~/my_workspace/tool/xxd.sh kernel.bin 1280 10
0000500: 55 89 E5 EB FE 47 43 43 3A 20                U....GCC:
[work@localhost kernel]$
```

▲图 5-39　可执行文件中的机器码

如图 5-39 所示，为了让大家放心，在这里我用 xxd.sh 脚本在文件 kernel.bin 偏移 1280 字节处查看了

10 字节的内容，其实前 5 字节就够啦，咱们前面已经判断出该代码段只有 5 字节大小、前 5 字节是 pushl%ebp 的机器码 55、movl %esp，%ebp 的机器码 89e5.jmp .L2 的机器码 ebfe。

由于前面第 2～5 个字节的值并不在字符编码的范围，所以只显示成'.'，虽然第一个字节 0x55 显示成了大写字符 U，但它可不是字符，人家 0x55 是 pushl %ebp 的机器码，显示区只是尽力按照可打印编码来显示，所以难免有些误导观众。另外，左边有 10 个字节，而右边的显示区只看到 9 个，原因是后面的字符串“GCC：”末尾还有个空格符。

问题似乎解决啦，估计您心里似乎有个疑问，pushl %ebp 的机器码 55 是我说的，不是我从哪里查出来的，怎么能证明以上的机器码就是对应这些汇编指令呢？为有这种想法的同学点赞，就冲您这严谨、一丝不苟的劲儿，绝对是做技术的料。最好的验证方法就是上虚拟机上实际跑一下，在虚拟机里面能够直接看出机器码。

做事就做彻底点吧，为了让大家相信，本来图 5-40 应该是下一节要讲的，现在提前给大家过目啦。

```
(0) [0x000000001500] 0008:c0001500 (unk. ctxt): push ebp                  ; 55
<bochs:3> n
Next at t=18053088
(0) [0x000000001501] 0008:c0001501 (unk. ctxt): mov ebp, esp              ; 89e5
<bochs:4> n
Next at t=18053089
(0) [0x000000001503] 0008:c0001503 (unk. ctxt): jmp .-2 (0xc0001503)      ; ebfe
```

▲图 5-40　汇编指令的机器码

图 5-40 所示最右边的那一列十六进制数字就是各行指令的机器码，指令“push ebp”对应的机器码是 0x55。

这下问题真地解决啦，在放心之余，我们还可以用 Linux 的命令 readelf 来确认一下，readelf 命令从名字上就能看出它的使命：读出 elf 文件的信息。为了让大家把重点信息一次看全，需要给 readelf 命令加个-e 参数，该参数的意义如图 5-41 所示。

-e 参数相当于'-h' + '-l' + '-S'，能让大家看到 elf header(file header)、program header 和 section header。

执行结果如图 5-42 所示。

```
[work@localhost kernel]$ readelf --help
Usage: readelf <option(s)> elf-file(s)
 Display information about the contents of ELF format files
 Options are:
  -a --all               Equivalent to: -h -l -S -s -r -d -V -A -I
  -h --file-header       Display the ELF file header
  -l --program-headers   Display the program headers
     --segments          An alias for --program-headers
  -S --section-headers   Display the sections' header
     --sections          An alias for --section-headers
  -g --section-groups    Display the section groups
  -t --section-details   Display the section details
  -e --headers           Equivalent to: -h -l -S
```

▲图 5-41　readelf 部分帮助

```
[work@localhost kernel]$ readelf -e kernel.bin
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0xc0001500
  Start of program headers:          52 (bytes into file)
  Start of section headers:          1372 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         6
  Section header string table index: 3

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        c0001500 000500 000005 00  AX  0   0  4
  [ 2] .comment          PROGBITS        00000000 000505 00002c 01  MS  0   0  1
  [ 3] .shstrtab         STRTAB          00000000 000531 00002a 00      0   0  1
  [ 4] .symtab           SYMTAB          00000000 00064c 000080 10      5   4  4
  [ 5] .strtab           STRTAB          00000000 0006cc 000025 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0xc0001000 0xc0001000 0x00505 0x00505 R E 0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4

 Section to Segment mapping:
  Segment Sections...
   00     .text
   01
[work@localhost kernel]$
```

▲图 5-42　readelf 输出

大家对照着前面的分析结果，在图 5-42 上对比一下吧，结果非常吻合（毕竟 xxd 命令和 readelf 命令

的输出都是从同一文件中读出来的，如果不吻合，肯定是 xxd 或 readelf 自己的问题）。由于咱们前面在讲述 section 和 segment 时有用过 readelf，当时有说过程序的输出，这里不再赘述啦，大家自己琢磨下吧。

本节到此就结束啦，有关 elf 格式的内容已经够用啦，到此也将告一段落。相信大伙儿经过理论+实践的学习过程已经理解了咱们所讲的部分，在下一节中，咱们要利用所学的 elf 知识将 kernel.bin 加载到内存中运行。

5.3.5　将内核载入内存

其实，我们等了这一刻好久好久，即使我不说，大家也有这样的认识，Linux 内核是用 c 语言写的，咱们肯定也要用 C 语言。其实……说点伤感情的话，今后的工作只是大部分（99%）都要用 C 语言来写，还有一些要用到汇编的地方。大家也不要因此心灰气馁（其实突然不用汇编还会想它呢，这不是玩笑），我在此过程中一定会尽我所能让内容简单易接受。

我们的内核文件是 kernel.bin，这个文件是由 loader 将其从硬盘上读出并加载到内存中的，到此，接力棒传到了最后一个选手的手里。也就是说，咱们需要事先把 kernel.bin 定入硬盘。好久不往虚拟硬盘上写东西了，甭说是大家，我都有点陌生了呢，不过好在操作很简单，写之前让我们先看看这块虚拟硬盘上的文件布局吧。

MBR 写在了硬盘的第 0 扇区，第 1 扇区是空着的，原因是个人喜好，其实不空着也行，不过硬盘那么大，何必搞得那么拥挤呢。因此 loader 写在硬盘的第 2 扇区，由于 loader.bin 目前的大小是 1342 字节，占用 3 个扇区，所以第 2～4 扇区不能再用啦，从第 5 扇区起我们可以自由使用。但此时我的强迫症又发作啦，我这里并没有接着第 5 扇区写，而是选的第 9 扇区（要是起始为 1 的话算是第 10 个扇区）。一是为了 loader 万一哪天要扩展，得预留出硬盘空间，二是您可能已经预计到了，隔开点显得更放心，这纯属是出于个人喜好做出的选择。

好，既然已经确定了写入扇区的位置，我们还是要通过 dd 命令往磁盘上写，命令如下。

dd if= kernel.bin of=/your_path/hd60M.img bs=512 count=200 seek=9 conv=notrunc 回车

seek 为 9，目的是跨过前 9 个扇区（第 0～8 个扇区），我们在第 9 个扇区写入。

count 为 200，目的是一次往参数 of 指定的文件中写入 200 个扇区。

至于为什么把 count 设成这么大，原因是这样的：每次写完内核后，咱们要往磁盘中同步内核文件，这样才能验证内核的正确性。按理说，咱们现在的内核文件不足 4 扇区，count=4 最合适。不过，内核发展越来越大时，每次都要根据实际内核文件大小去改写 count 参数，这样就难免会有忘记修改的情况。之前我就深受其苦，内核文件变大了，而 count 忘记调整，造成写入硬盘中的内核文件不完整，所以到后来，程序运行不受控制，以至于调试的时候都调晕啦，看着 CPU 中跑的指令我完全蒙圈了，根本不是自己写的。恍然大悟之后，我就干脆一步到位，因为我们将来的内核大小不会超过 100KB，所以直接把 count 改为 200 块扇区。另外请大家不用担心，dd 命令会自己判断写入的数据量，如果参数 if 指定的文件体积小于 count*bs，只按实际文件大小写入。

不过，估计您也觉得参数太多了，为了方便，我通常是把下面三个命令，编译、链接、再写入硬盘一起完成，您可以将它们写成一个脚本，脚本内容如下。

gcc -c -o main.o main.c && ldmain.o -Ttext 0xc0001500 -e main -o kernel.bin && dd if=kernel.bin of=/your_path/hd60M.img bs=512 count=200 seek=9 conv=notrunc

好啦，上面命令在回车之后，我们的内核文件就成功写进磁盘了。

菜配好啦，就等下锅啦，我们的内核是由 loader 加载的，所以我们还要去修改下 loader.S。

loader.S 需要修改两个地方。

- 加载内核：需要把内核文件加载到内存缓冲区。
- 初始化内核：需要在分页后，将加载进来的 elf 内核文件安置到相应的虚拟内存地址，然后跳过去执行，从此 loader 的工作结束。

先说第一个加载内核，这里所说的加载内核只是把内核从硬盘上拷贝到内存中，并不是运行内核代码。这项工作在开启分页前后都可以，不过为了简单，咱们把它安排在分页开启之前加载。

话说内核加载到内存中，得有个加载地址，也就是缓冲区。其实开发经验少的同学对缓冲区这个概念总是觉得有点“只可意会不可言传”的意思。借此机会多说两句。缓冲区，buffer，意味存放物品的地点，也就是用于加工处理中暂存数据的地方。生活中的缓冲区例子有很多，比如水杯是水的缓冲区，水不是直

接入口的，总有个中间载体作为中转，然后才入口。而且，水杯的作用相当于暖瓶或水房的缓存，咱们不是喝一口水就跑到水房接一口水，而是一次接一大杯，回来慢慢喝，这样就减少了去水房的次数。由此可见，缓冲区，既有存放数据的空间之意，又有提高效率的缓存之意。换在计算机世界里，缓冲区必然也是个能存储数据的介质，比如咱们这里所说的内存。

好啦，不能扯太远啦，咱们的缓冲区设在哪里呢？这不是乱放的，得参考下目前内存中哪个地方还有可用的空间，千万不能覆盖了重要数据。也许大家首先想到的是很久之前说到的那个内存布局图，赞，答对啦，不过，大家不用往前翻看啦，一向体贴的我已经将其重点部分摘到这里啦，大家请看图 5-43。

同样是在很久很久之前，我曾经告诉过大家，咱们的内核很小，所以只需要在低端 1MB 中安身就够啦。可是这 1MB 并不完全都是咱们的，还有好多重要数据在这里呢，所以咱们要在这 1MB 中再找个一亩三分地给内核。

9FC00	9FFFF	1K	EBDA（Extended BIOS Data Area)扩展 bios 数据区
✓ 7E00	9FBFF	622080 B 约 608K	可用区域
✓ 7C00	7DFF	512B	MBR 被 BIOS 加载到此处，共 512 字节
✓ 500	7BFF	30464B 约 30K	可用区域
400	4FF	256B	BIOS Data Area（BIOS 数据区)
000	3FF	1K	Interrupt Vector Table（中断向量表)

▲图 5-43　低端 1MB 中可用内存

在图 5-43 中可用的部分，我用三个勾给标出来了。中间的那个勾似乎有点不近人情，人家 MBR 刚刚结束使命，这就要被覆盖啦……确实，如果不覆盖，这 512 字节就要把这连续可用的区域隔开啦，这会儿不用它，将来也要用到。

内核被加载到内存后，loader 还要通过分析其 elf 结构将其展开到新的位置，所以说，内核在内存中有两份拷贝，一份是 elf 格式的原文件 kernel.bin，另一份是 loader 解析 elf 格式的 kernel.bin 后在内存中生成的内核映像（也就是将程序中的各种段 segment 复制到内存后的程序体），这个映像才是真正运行的内核。

将来内核肯定是越来越大，为了多预留出生长空间，咱们要将内核文件 kernel.bin 加载到地址较高的空间，而内核映像要放置到较低的地址。内核文件经过 loader 解析后就没用啦，这样内核映像将来往高地址处扩展时，也可以覆盖原来的内核文件 kernel.bin。所以咱们的结论是在 0x7e00～0x9fbff 这片区域的高地址中找一亩地给 kernel.bin，这里我擅自做主啦，帮大家选的是 0x70000。为什么？没有为什么，随意选的，取了个整而已，就是觉得 0x70000～0x9fbff 有 0x2fbff=190KB 字节的空间，而我们的内核不超过 100KB，够用就行。

好，万事俱备啦，代码走起，请大家过目代码 5-7。

代码 5-7 （project/c5/c/boot/loader.S）

```
147 ; -------------------------   加载 kernel  ----------------------
148    mov eax, KERNEL_START_SECTOR  ; kernel.bin 所在的扇区号
149    mov ebx, KERNEL_BIN_BASE_ADDR
       ; 从磁盘读出后，写入到 ebx 指定的地址
150    mov ecx, 200                  ; 读入的扇区数
151
152    call rd_disk_m_32
153
154    ; 创建页目录及页表并初始化页内存位图
155    call setup_page
```

代码 5-7 属于 loader 的一部分，它的作用是把内核文件从硬盘上加载到内存中，下面简要说一下。

第 148～149 行的 KERNEL_START_SECTOR 和 KERNEL_BIN_BASE_ADDR 在 boot/include/ boot.inc 中定义，其值分别为 0x9 和 0x70000。

第 150 行的 ecx 为 200，这是读入的扇区数，这里应该同前面用 dd 命令往硬盘上写入内核文件时的参数 count 保持一致，原因你懂的，不解释。

以上的 eax、ebx、ecx 是函数 rd_disk_m_32 的三个参数，为调用下面的函数做准备。

第 152 行的函数是 rd_disk_m_32，用于从硬盘上读取文件。它的三个参数已经在上面赋值了。由于目前已经在 32 位保护模式下，所以相比之前位于 mbr 中的函数 rd_disk_m_16，rd_disk_m_32 只是版本由 16 位变成了 32 位的，函数实现原理相差无几，主要体现在里面所用的寄存器变成了 32 位。所以，就不细说啦，大家一看就明白啦。

接下来的第 155 行就开始创建页表啦，把它放在这是为了让大家知道代码 5-7 是加到了哪里，承上启

下。setup_page 函数实现没变，无需多说。

内核加载到缓冲区中后，现在该说要修改的第二处啦，也就是初始化内核。

内核文件 kernel.bin 是 elf 格式的二进制可执行文件，初始化内核就是根据 elf 规范将内核文件中的段（segment）展开到（复制到）内存中的相应位置。在分页模式下，程序是靠虚拟地址来运行的，无论是内核，还是用户程序，它们对 CPU 来说都是指令或数据，没什么区别，交给 CPU 的指令或数据的地址一律被认为是虚拟地址。坦白说，内核文件中的地址是在编译阶段确定的，里面都是虚拟地址，程序也是靠这些虚拟地址来运行的。但这些虚拟地址实际上是我们在初始化内核阶段规划好的，即想安排内核在哪片虚拟内存中，就将内核地址编译成对应的虚拟地址。而目前我们初始化的是内核，它在物理低端 1MB 内存中，初始化工作取决于这 1MB 物理内存中哪块空间可用，所以，现在还要看图 5-43，从中找块合适的内存空间来容纳内核映像。

其实大家早已经知道内核的入口虚拟地址是 0xc0001500 啦。但现在大家要假装不知道，配合一下啊，咱们说一下 0xc0001500 是怎么来的。

物理内存中 0x900 处是 loader.bin 加载的地址，在 loader.bin 的开始部分是 GDT，它可是必须要保留下来的，可不能覆盖，我们不打算在内核中重新定义它，以后都要指望它了。虽然 loader 的工作结束啦，但 loader 所完成的工作成果咱们还得继续发扬，继续用。预计 loader.bin 的大小不会超过 2000 字节。所以咱们可选的起始物理地址是 0x900+2000=0x10d0（不要把注意力放在这个奇怪的数上，偶然得出的）。内存很大，但也尽量往低了选，于是凑了个整数，选了 0x1500 作为内核映像的入口地址。

根据咱们的页表，低端 1MB 的虚拟内存与物理内存是一一对应的，所以物理地址是 0x1500，对应的虚拟地址是 0xc0001500。这就解释了在 5.3.1 节中，链接命令 ld 中用-Ttext 指定了代码段的起始虚拟地址，再把命令搬过来给大家看下。

```
ld kernel/main.o -Ttext 0xc0001500 -e main -o kernel/kernel.bin
```

好，现在咱们得说一下初始化内核的代码，见代码 5-8。

代码 5-8　（project/c5/c/boot/loader.S）

```
193 ;----------   将 kernel.bin 中的 segment 拷贝到编译的地址   -----------
194 kernel_init:
195     xor eax, eax
196     xor ebx, ebx        ;ebx 记录程序头表地址
197     xor ecx, ecx        ;cx 记录程序头表中的 program header 数量
198     xor edx, edx        ;dx 记录 program header 尺寸，即 e_phentsize
199
200     mov dx, [KERNEL_BIN_BASE_ADDR + 42]
; 偏移文件 42 字节处的属性是 e_phentsize，表示 program header 大小
201     mov ebx, [KERNEL_BIN_BASE_ADDR + 28]
        ; 偏移文件开始部分 28 字节的地方是 e_phoff
        ;表示第 1 个 program header 在文件中的偏移量
202     ; 其实该值是 0x34，不过还是谨慎一点，这里来读取实际值

203     add ebx, KERNEL_BIN_BASE_ADDR
204     mov cx, [KERNEL_BIN_BASE_ADDR + 44]
; 偏移文件开始部分 44 字节的地方是 e_phnum，表示有几个 program header

205 .each_segment:
206     cmp byte [ebx + 0], PT_NULL
; 若 p_type 等于 PT_NULL，说明此 program header 未使用
207     je .PTNULL
208
209     ;为函数 memcpy 压入参数，参数是从右往左依然压入
        ;函数原型类似于 memcpy（dst，src，size）
210     push dword [ebx + 16]
; program header 中偏移 16 字节的地方是 p_filesz
        ;压入函数 memcpy 的第三个参数：size

211     mov eax, [ebx + 4]      ; 距程序头偏移量为 4 字节的位置是 p_offset
212     add eax, KERNEL_BIN_BASE_ADDR
        ; 加上 kernel.bin 被加载到的物理地址，eax 为该段的物理地址
```

```
213     push eax                ; 压入函数 memcpy 的第二个参数：源地址
214     push dword [ebx + 8]    ; 压入函数 memcpy 的第一个参数：目的地址
                    ; 偏移程序头 8 字节的位置是 p_vaddr，这就是目的地址
215     call mem_cpy            ; 调用 mem_cpy 完成段复制
216     add esp, 12             ; 清理栈中压入的三个参数
217 .PTNULL:
218     add ebx,  edx           ; edx 为 program header 大小，即 e_phentsize
                                ;在此 ebx 指向下一个 program header
219     loop .each_segment
220     ret
221
222 ;----------  逐字节拷贝 mem_cpy（dst, src, size） ------------
223 ;输入:栈中三个参数（dst, src, size）
224 ;输出:无
225 ;---------------------------------------------------------
226 mem_cpy:
227    cld
228    push ebp
229    mov ebp,  esp
230    push ecx       ; rep 指令用到了 ecx
                      ; 但 ecx 对于外层段的循环还有用，故先入栈备份
231    mov edi, [ebp + 8]      ; dst
232    mov esi, [ebp + 12]     ; src
233    mov ecx, [ebp + 16]     ; size
234    rep movsb               ;  逐字节拷贝
235
236    ;恢复环境
237    pop ecx
238    pop ebp
239    ret
```

对于可执行程序，我们只对其中的段（segment）感兴趣，它们才是程序运行的实质指令和数据的所在地，所以我们要找出程序中所有的段。

函数 kernel_init 的作用是将 kernel.bin 中的段（segment）拷贝到各段自己被编译的虚拟地址处，将这些段单独提取到内存中，这就是平时所说的内存中的程序映像。kernel_init 的原理是分析程序中的每个段（segment），如果段类型不是 PT_NULL（空程序类型），就将该段拷贝到编译的地址中。

现在内核已经被加载到 KERNEL_BIN_BASE_ADDR 地址处，该处是文件头 elf_header。在我们的程序中，遍历段的方式是指向第一个程序头后，每次增加一个段头的大小，即 e_phentsize。该属性位于偏移程序开头 42 字节处。为了以后遍历段时方便，避免了频繁的访问内存，在第 200 行，我们用寄存器 dx 来存储段头大小，这样，每遍历一个段头时，就直接从 dx 中获取段头大小，这将在第 218 行体现。

为了找到程序中所有的段，必须要获取程序头表。在文件开头偏移 28 字节处是属性 e_phoff，该属性表示程序头表在文件中的偏移量，程序头表是程序头 program header 的数组，所以 e_phoff 也就是第 1 个 program header 在文件中的偏移量。第 201 行，在内存 e_phoff 处取值，将得到的程序头表偏移量存入寄存器 ebx。

我们需要的是程序头表的物理地址，由于此时的 ebx 还是程序头表文件内的偏移量，所以要将其加上内核的加载地址，这样才是程序头表的物理地址。所以在第 203 行为 ebx 加上了内核文件的加载地址 KERNEL_BIN_BASE_ADDR。最终 ebx 寄存器作为程序头表的基址，用它来遍历每一个段，此时 ebx 指向程序中的第 1 个 program header。

我们已经知道，段是由程序头（program header）来描述的，一个程序头代表一个段。在知道了第一个程序头的地址后，为了遍历所有的程序头，还需要知道程序中程序头的数量，也就是段的数量，这是由 elf_header 中的属性 e_phnum 决定的，它在 elf_header 中偏移为 44。我们通常用 cx 寄存器来做循环计数器，所以在第 204 行，汇编语句“mov cx, [KERNEL_BIN_BASE_ADDR + 44]”将段的数量赋值给寄存器 cx。

现在程序头表地址在寄存器 ebx 中，而且又知道了程序头表中段的数量，所以现在可以遍历每一个段的信息啦，其工作在代码第 205～220 行中完成。

在第 206 行，程序先判断下段的类型是不是 PT_NULL，PT_NULL 是在 boot/include/boot.inc 中定义的宏，其值为 0，该意义表示空段类型。（PT_NULL 也可以在 Linux 系统的/usr/include/elf.h 中找到其定义：#define PT_NULL 0。）

在第 207 行，如果发现该段是空段类型的话，就跨过该段不处理，跳到.PTNULL 处，也就是第 217 行。

指定下一个段是通过在程序头表地址处加上一个段的大小 e_phentsize 来实现的，e_phentsize 的值咱们已经将其存储在 dx 寄存器啦，所以在第 218 行，直接将 ebx，也就是当前 program header 地址，加上 edx，ebx 便指向了下一个段的 program header。edx 的高 16 位为 0，所以这里用 add ebx，　edx 没有问题。

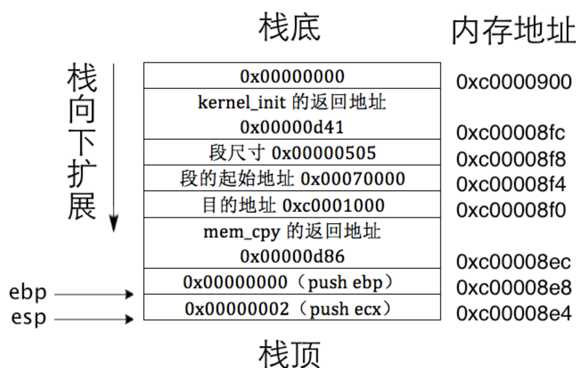
第 209～216 行，程序中的段通过 mem_cpy 函数复制到段自身的虚拟地址处。在这里，我们涉及到了函数调用约定的知识，不过为了叙述得更清楚，在这里我不想简单地说，在下一章中我们专门拿出一节来说这事儿。在此我还是本着够用的原则，把用到的部分给您说明白。

我们在此实现的函数是 mem_cpy，不是 c 标准库中的 memcpy 函数，将来我们会在内核中实现 memcpy。memcpy 原型是 void *memcpy(void *dest, const void *src, size_t n)，功能是将 src 指向的地址空间处的连续 n 个字节拷贝到 dest 指向的地址空间。我们得学习它的用法，在汇编语言中用 mem_cpy 函数实现了它，此函数的原型相当于 mem_cpy(void* dst, void* src, int size)。所以我们也要提供三个参数才能使用它。这三个参数都在程序头 program header 中，所以它们都可以基于 ebx 再增加适当的偏移量来得到。大家结合图 5-36 所示的 program header 结构，很容易理解第 210～214 行的代码。

第 215 行是调用 mem_cpy，这涉及到为该函数传入参数的问题。在汇编语言中传递参数的方法太多了，原因是汇编语言太灵活了，不怎么受约束，咱们可以访问到的资源太多了。所以，主调函数可以把参数放在寄存器中，也可以放在栈中，而栈就是内存，所以只要大家高兴，也可以把参数直接放到某块内存中，类似共享内存的方式来传递参数。主调函数以上面任意一种方式传递参数，被调函数都可以轻松地拿到参数。

在这里，我们把参数放到了栈中保存，大家注意到了，参数入栈的顺序是先从最右边的开始，最后压入参数最左边的，其实这是某种约定，要不，为什么不先把中间的参数 src 入栈呢？既然主调函数按照从右到左的顺序在栈中压入参数，被调函数中必须分清楚这三个参数分别在栈中哪个位置。栈是向下扩展的，这一点通过 push 指令压栈时，栈指针 esp 的值越来越小能体现出来，所以最后压入的第 1 个参数离栈顶（esp 指向的地址）最近，最先入栈的第 3 个参数离栈顶最远。我们来看下在参数入栈后并调用函数时，栈中布局是什么，还是拿 call mem_cpy 为例，如图 5-44 所示。

先说点题外的，如图 5-44 所示，我们都知道，栈是从上往下发展的，但很少有同学意识到栈底是用不上的。从图中可见，地址 0xc0000900 处的值为 0，这并不是咱们压入的值。push 操作的原理是先进行 sub esp，4，再 mov dword [esp]，操作数，所以栈底处是空的，您懂的。



▲图 5-44　函数调用栈中信息

由于栈指针 esp 已经在 loader.S 中被加上了 0xc0000000，所以其栈中地址都是内核所在的 0xc0000000 以上的高地址。用 call 指令进行函数调用时，CPU 会自动在栈中压入返回地址，由图可见，当调用 kernel_init 函数时，当时的栈指针是 0xc00008fc，所以 kernel_init 的返回地址被存储在 0xc00008fc 处。栈中地址 0xc00008f8 处的内容是提供给函数 mem_cpy 的第三个参数，即 size。地址较低的 0xc00008f4 处是它的第二个参数，即 src 地址，0xc00008f0 处是它的第一个参数，即 dst。

在 mem_cpy 的实现中，我们访问栈中的参数是基于 ebp 来访问的，这通常意味着要将 esp 的值赋给 ebp。由于不知道 ebp 中的值是不是重要，好的习惯是提前将 ebp 备份起来，这就是在第 228 行的目的，将 ebp 入栈备份，这样在函数结束时能够将其恢复。我们在第 229 行将 esp 赋值给了 ebp。所以在图 5-35 中标出了 ebp 的指向，由于后来在第 230 行又将 ecx 入栈，故 esp 已经小于 ebp。

栈中每个单元占用 4 字节，既然是基于 ebp 来获得栈中的参数，那么如图所示，第 1 个参数 dst 的地址是 ebp+8，第 2 个参数 src 的地址是 ebp+12，第 3 个参数 size 的地址是 ebp+16。分别对这些地址用中括号取值后，便可以得到实际的参数。

在继续往下说之前，要给大家介绍个数据复制小团队。

首先要说一下字符串“搬运”指令族：movsb、movsw、movsd。其中的 movs 代表 move string，后面的 b 代表 byte，w 代表 word，d 代表 dword。所以 movsb 的功能是搬运（复制）1 字节，movsw 的功能是搬运（复制）2 字节，movsd 的功能是搬运（复制）4 字节。数据从哪里来，搬到哪里去呢？这三条指令是将 DS：[E]SI 指向的地址处的 1、2 或 4 个字节搬到 ES：[E]DI 指向的地址处，16 位环境下源地址指针用 SI 寄存器，目的地址指针用 DI 寄存器，32 位环境下源地址则用 ESI，目的地址则用 EDI。话说虽然这三个指令叫字符串指令，但它们可不是只用在字符串上，因为字符串中的字符不也是按字节来存储的吗，任何数据在内存中都以字节存储单元来访问，字符串只是表象，本质上是复制字节，所以它更多地被通用于复制数据。

以上三个命令只是复制固定的字节数，每执行一次就复制 1 字节、2 字节或 4 字节，如果大量的数据需要复制，则需要连续的运行，所以要介绍另外一个指令 rep。

rep 指令是 repeat 重复的意思，该指令是按照 ecx 寄存器中指定的次数重复执行后面的指定的指令，每执行一次，ecx 自减 1，直到 ecx 等于 0 时为止，所以在用 rep 重复执行某个指令之前，一定要将 ecx 寄存器提前赋值。

似乎说完了，但其实还差点什么，您想，如果想要复制一大块数据的话，总该有人更新数据的来源和目的地吧。movs[bwd]只是从[e]si 指向的地址处搬运 1、2、4 字节到[e]di 指向的地址处，它不会自动更新[e]si 和[e]di。咱们总不能翻来覆去从同一个源地址搬运数据到另一个相同的目的地址吧。所以，cld 和 sld 指令就派上用场了，这两个指令本质上是控制重复执行字符串指令时的[e]si 和[e]di 的递增方式，递增方式是指它们的值逐渐变大，还是逐渐变小，也就是说，地址是往高地址方向变化，还是往低地址方向变化，这就是所说的方向。cld 是指 clean direction，该指令是将 eflags 寄存器中的方向标志位 DF 置为 0，这样 rep 在循环执行后面的字符串指令时，[e]si 和[e]di 根据使用的字符串搬运指令，自动加上所搬运数据的字节大小，这是由 CPU 自动完成的，不用人工干预。比如，执行一次 movsd，[e]si 和[e]di 就自动加 4，执行一次 movsb，[e]si 和[e]di 就自动加 1。有清除方向标志位就会有设置方向标志位，std 是 set direction，该指令是将方向标志位 DF 置为 1，每次 rep 循环执行后面字符串指令时，[e]si 和[e]di 自动减去所搬运数据的字节大小。

也许 CPU 认为地址由低向高处发展是理所应当的，这无需设置，所以此时 DF 标志为 0。当由高地址向低地址发展时，这不是正常自然的现象，所以需要强调一下，故要将 DF 标志置为 1。

注意，并不是在任何字符串控制指令中[e]si 和[e]di 都同时增减，这要看字符串操作指令是否都用到了它们，处理器只会增加用到的那个。字符串操作指令有很多，比如有 movs[bwd]、ins[bwd]和 outs[bwd]、lods[bwd]和 stos[bwd]，esi 和 edi 并不是被以上三组指令同时使用的，只有 movs[bwd]才同时使用 esi 和 edi，通过 rep 指令组合执行时，esi 和 edi 根据 DF 位的值自增或自减。ins[bwd]从端口读入数据到内存的目的地址，故只涉及到 edi 的自增自减。outs[bwd]把内存中的源数据写入端口，故只涉及到 esi 的自增自减。lods[bwd]把内存中的源数据加载到寄存器 al、ax 或 eax，自增自减操作也只涉及 esi。而 stos[bwd]将 al、ax、eax 中的值写入到内存中的目的地址，故也只涉及 edi 的自增自减。

好啦，在稍微扩展了一小下之后，咱们回到正题。

有了 movs[bdw]指令族、重复执行指令 rep、方向指令 cld 和 std，这三剑客在一起配合工作就能够自由复制任何大块数据啦。万事俱备，回到正题。

第 227 行的 cld 指令其实放在 movsb 之前就行，它用于清除方向标志，让数据的源地址和目的地址逐渐增大。

由于外层函数也要用 ecx 作为遍历段的循环计数，所以您明白了，这里的第 230 行为什么要将 ecx 入栈备份啦，这样在 ecx 用完之后，在 mem_cpy 执行结束前通过 pop 指令将 ecx 和 ebp 恢复，以便外层遍历段的循环中保持 ecx 正确。

在第 231～233 行，为复制工作所需要的条件初始化，esi 和 edi 指向了要复制的段的来源地址和目的地址，ecx 是为 rep 指令做准备的，指定了调用 movsb 指令的次数。在此提醒一下，段寄存器 DS 和 ES 在进入保护模式之初就被赋成相同的选择子了，它们都指向同一个段描述符，故它们在此工作正确，请大伙儿放心。

一切就绪之后，在第 234 行，rep movsb，这三剑客团队就开始合作啦。

mem_cpy 返回后，程序流程回到第 216 行，这是清理在调用 mem_cpy 之前在栈中压入的 size、src、

dst，这三个参数共占 3*4=12 字节，所以将 esp 加上 12，于是栈顶跨过了它们，这三个参数所占的空间可被其他压栈操作覆盖。

每个函数中都要有个返回指令，这里用的是 ret 指令，以后我们还会接触到其他返回指令。之前在用 call 指令调用函数时，无论是调用 kernel_init，还是 mem_cpy，CPU 都会将函数的返回地址压入栈中保存，这是为函数体中的 ret 指令准备的，换句话说函数不会自己返回，是通过 ret 来返回的。ret 指令将栈顶中的值作为返回地址，所以，一定要确保在调用 ret 时，位于栈顶处的数据是正确的返回地址。一般情况下，我们在函数体中保证 push 操作和 pop 操作配套成对，正如在 mem_cpy 的实现中，有两个 push 入栈操作，在函数返回前就要有两个 pop 出栈操作。

咱们的函数中用的都是 ret 近返回指令，所以只会在栈顶弹出 4 字节的数据作为代码段的偏移地址为 EIP 寄存器赋值，从而恢复了程序执行流。

介绍完内核初始化的函数 kernel_init 后，本节代码部分还差一点点没说啦，下面看代码 5-9。

代码 5-9　（project/c5/c/boot/loader.Ss）

```
…略
179     ;在开启分页后,用 gdt 新的地址重新加载
180     lgdt [gdt_ptr]                       ; 重新加载
181
182 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  此时不刷新流水线也没问题  ;;;;;;;;;;;;;;;;;;;;;;;;
183 ;由于一直处在 32 位下，原则上不需要强制刷新
    ;经过实际测试没有以下这两句也没问题
184 ;但以防万一，还是加上啦，免得将来出来莫名其妙的问题
185     jmp SELECTOR_CODE:enter_kernel      ;强制刷新流水线，更新 gdt
186 enter_kernel:
187 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
188     call kernel_init
189     mov esp, 0xc009f000
190     jmp KERNEL_ENTRY_POINT               ; 用地址 0x1500 访问测试，结果 ok
…略
```

在代码的开头是咱们之前已经完成的重新加载 GDT，就是将原来 GDT 的基地址 0x900 变成 0xc0000900 后重新加载。按理说，当前已经是 32 位环境啦，而且内核也是 32 位程序，不需要“显式”地清空流水线。实话实说，之前 loader.S 中是没有第 185～186 行的，而且经我简单测试后其运行结果也是正确的。不过，在调试过程中有可能会碰到稀奇古怪的问题，当然这绝对是人为的错误，不要轻易怀疑计算机。对于内核中这类“灵异”事件，咱们当然希望少碰到，某些 bug 真地会让人调试好多天，所以为了保险起见，还是用无条件跳转指令刷新了流水线，请大家知晓。

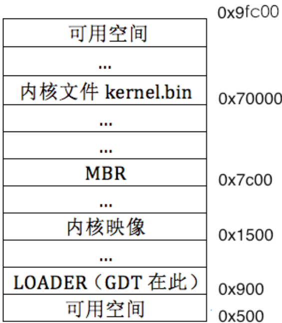
在进入内核之后，我们用的栈也要重新规划了，栈起始地址不能再用之前的 0xc0000900 啦。为了方便编写程序，我们在进入内核前将栈指针改成我们期待的值，在第 189 行，我们将 esp 改成了 0xc009f000。此地址的选择也是根据图 5-43。也许有同学会说，为什么不把 esp 选为 0x9fc00，这才是最合理的。没错，您说得对，我们都是会过日子的人，0x9fc00 确实是最省空间的选择，这样做，以后的程序也不会出错。但这牵扯到以后要说的 pcb，即程序控制块（咱们在以后线程相关章节会细说 pcb，这里仅要求大家对此有个浅表的了解即可），每个 pcb 都是自然页，也就是要求 4KB 对齐，即 4KB 的范围是 0x000～0xfff，而不是类似 0x333～0x1332 这样的范围。我们打算将在 4KB 内的最高地址作为栈底，如果以 0x9fc00 作为栈底，虽然不出会什么问题，但它显得太个性了，比其他 pcb 少了 0x400 字节。所以，为了统一 pcb 大小，我们这里选择栈底的要求是：它接近最大可用地址 0x9fbff，并且以 4KB 对齐，所以 0x9f000 是最合适的。

为了打消部分同学的疑虑，容我再多说两句。我担心有同学可能会这样想，咱们 loader 加载的物理地址是 0x900，loader 中使用的栈的栈底是 0x900，栈是往下发展的，在 loader 以后的压栈操作中，并不会破坏掉 loader 自身，似乎这种“完美”的方案可以在咱们的 kernel 中延续，也就是为何不让 kernel 的栈为入口地址之下？比如咱们这里的入口地址是 0xc0001500，也让栈底 esp 为该值有何不妥。不管怎么说，如果有这种想法，说明您是个爱动脑的同学，我会为您悄悄点赞。其实 loader.bin 是纯二进制文件，而 kernel 是 elf 格式的二进制文件，这两者的区别是 elf 比纯二进制文件多了文件头，纯二进制文件相当于 elf 文件中的所有段（segment）的集合。在前面我们分析过啦，程序的入口地址是很可能会在段中的，并不是在

段的起始，就拿咱们的 kernel.bin 来说，代码段的入口地址是 0xc0001500，起始地址却是 0xc0001000，入口并不在段的开头。其实 0xc0001000～0xc0001500 之间的部分是文件头，并不是真正的代码。真要把 esp 赋值为 0xc0001500，如您所料，将来内核中有压栈操作时一定会破坏 0xc0001000～0xc0001500 之间的部分，虽然它只是文件头，不是实际代码，似乎破坏了也无关紧要，但这样做确实不美啊。您看，咱们的内核预计 70KB 左右，起始物理地址是 0x1500，而栈底若为 0x9f000，这 0x9f000～0x1500 约为 630KB 的空间，在正常情况下，栈是不会碰撞到内核的，这样多省心。所以，兄弟们，我看还是算了吧，咱就选个高地址 0x9f000，就它了。

loader 通过第 190 行的跳转指令进入内核。这里所见的 KERNEL_ENTRY_POINT 是 boot/include/boot.inc 中定义的宏，其值为 0xc0001500，它正是我们用 ld 命令链接 kernel 时指定的代码段地址，这个宏必须要与其一致才行。

经过这样一番的规划后，现在 0x500～0x9fbff 可用内存中，咱们自己的文件布局如图 5-45 所示。



▲图 5-45 程序布局

和内核相关的内容咱们暂告一段落，在本章的结束，咱们说说保护模式下最闪亮的内容——特权。

5.4 特权级深入浅出

所谓保护模式下的“保护”，主要体现在特权级上，以后随着后面工作的展开，会越来越多地和它们打交道。

保护模式的安全性也体现了“特权”：为了维护计算机世界的“和平”，避免潜在的危险，对于那些不受控的程序，剥夺它们的部分能力，使它们没有杀伤力，让它们只能老老实实地运行。

5.4.1 特权级那点事

先给大家笼统地介绍下特权级那点事。

整个计算机世界其实可以分为两部分，访问者和受访者。访问者是动态的，具有能动性，它主动去访问各种资源。受访者是静态的，它就是被访问的资源，只能干坐着等待访问者光顾。访问者的特权级可以变，受访者的特权不能变。

拿开车举例，CPU 相当于汽车，驾驶车的人可以是普通人，也可以是警察。同样一辆车，只有警察才能把车开到警局，普通人开着这辆车去警局会被拦下的，在警局门口的警卫说了，只有警车才能开进警局，警卫判断汽车是否为警车的标准，检查司机是否为警察，只要是警察开的车，一律按警车放行处理。当前特权级就是指 CPU 的状态，普通人的特权为 3，警察的特权为 0。当普通人想把车开进警局，到警局门口就得换一位身份为警察的司机，这就是特权级变换，而 CPU，也就是这辆车，它在硬件上始终是不变的，车还是那辆车，只是车的角色在变。开车的人不同，车的角色就不同，普通人开，这车就是普通的私家车，当警察作为司机时，它就成了警车，到了警局门口，警卫便让其通行。

建立特权机制是为了通过特权来检查合法性，整个计算机世界的特权检查，都是发生在“访问者”在访问“受访者”的一刹那，实际上就是检查访问者的特权级和受访者的特权级是否匹配。

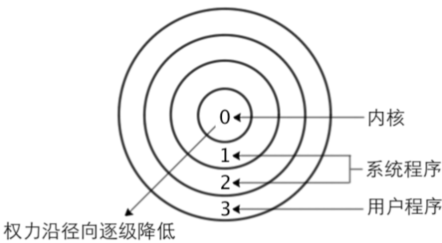
不知道各位看官听我说这个例子后有没有对特权有个概貌的认识，下面咱们从细节上展开讨论。

CPU 既是大脑，又是警察，它负责维护计算机内的安全。它将程序拥有的权利分为 4 个等级，这就是保护模式下特权级的由来。

特权级按照权力从大到小分为 0、1、2、3 级，没错，数字越小，权力越大，0 级特权能力最大，3 级特权能力最小。

0 级特权是我们操作系统内核所在的特权级，必须得让操作系统处于至高无上的地位，这样它的子民（应用程序）才不会反了天。计算机在启动之初就以 0 级特权运行，MBR 是咱们所写的第一个程序，它是

含着金钥匙出生的，自从它从 BIOS 那里接过第一棒的时候，它已经是像神一样处于 0 级特权了。整个系统的特权级分布如图 5-46 所示。



▲图 5-46　特权级

特权级是人为设计的，各层有各层的使命。

如图 5-46 所示，特权这样分级的思想是计算机是由操作系统来掌控的，操作系统只相信自己的代码，自己的特权最高，越是别人的程序越不放心，所以，不放心的程序特权就低。

操作系统位于最内环的 0 级特权，它要直接控制硬件，掌控各种核心数据，所以它的权利必须最大。系统程序分别位于 1 级特权和 2 级特权，运行在这两层的程序一般是虚拟机、驱动程序等系统服务。在最外层的是 3 级特权，我们的用户程序就运行在此层，用户程序被设计为“有需求时找操作系统”，所以它不需要太大的能力，能完成一般工作即可，因此它的权利最弱。

好啦，特权级从大体上就这点儿事，本节到这结束，下节再往细节上说说。

5.4.2　TSS 简介

本想着将来在介绍用户进程时再讨论 TSS，但本节中所讲的特权级与它有着密不可分的联系，TSS 作用不只涉及特权级，还包括任务寄存器环境。任务管理相关的内容，为了不干扰大家，这里只介绍和特权级相关的内容，待后面用到更多内容时再和读者细说。

TSS，即 Task State Segment，意为任务状态段，它是处理器在硬件上原生支持多任务的一种实现方式，也就是说处理器原本是想让操作系统开发厂商利用此结构实现多任务的，人家处理器厂商已经提供了多任务管理的解决方案，尽管后来操作系统并不买账，这是后话，以后再议。TSS 是一种数据结构，它用于存储任务的环境。咱们一睹为快，见图 5-47。

31	15　　　0	
I/O位图在TSS中的偏移地址	（保留）　T	100
（保留）	ldt 选择子	96
（保留）	gs	92
（保留）	fs	88
（保留）	ds	84
（保留）	ss	80
（保留）	cs	76
（保留）	es	72
edi		68
esi		64
ebp		60
esp		56
ebx		52
edx		48
ecx		44
eax		40
eflags		36
eip		32
cr3(pdbr)		28
（保留）	SS2	24
esp 2		20
（保留）	SS1	16
esp1		12
（保留）	SS0	8
esp 0		4
（保留）	上一个任务的 TSS 指针	0

32位TSS结构

▲图 5-47　32 位 TSS 结构

TSS 是每个任务都有的结构，它用于一个任务的标识，相当于任务的身份证，程序拥有此结构才能运行，这是处理器硬件上用于任务管理的系统结构，处理器能够识别其中每一个字段。该结构看上去也有点复杂，里面众多寄存器都囊括到这 104 字节中啦，其实这 104 字节只是 TSS 的最小尺寸，根据需要，还可以再接上个 IO 位图，这些内容将在后面章节用到时补充。这里目前只需要关注 28 字节之下的部分，这里包括了 3 个栈指针，这是怎么回事呢。

在没有操作系统的情况下，可以认为进程就是任务，任务就是一段在处理器上运行的程序，相当于某个计算机高手在脱离操作系统的情况下所写的代码，它能让计算机很好地运行。在有了操作系统之后，程序可分为用户程序和操作系统内核程序，故，之前完整的一个任务也因此被分为用户部分和内核部分。由于内核程序位于 0 特权级，用户程序位于 3 特权级，所以，一个任务按特权级来划分的话，实质上是被分成了 3 特权级的用户程序和 0 特权级的内核程序，这两部分加在一起才是能让处理器完整运行的程序，也就是说完整的任务要历经这两种特权的变换。所以，我们平时在 Linux 下所写的程序只是个半成品，咱们只负责完成用户态下的部分，内核态的部分由操作系统提供。

任务是由处理器执行的，任务在特权级变换时，本质上是处理器的当前特权级在变换，由一个特权级变成了另外一个特权级。这就开始涉及栈的问题了，处理器固定，处理器在不同特权级下，应该用不同特权级的栈，原因是如果在同一个栈中容纳所有特权级的数据时，这种交叉引用会使栈变得非常混乱，并且，用一个栈容纳多个特权级下的数据，栈容量有限，这很容易溢出。举个例子，处理器位于 0 特权级时要用 0 特权级的栈，3 特权级下也只能用 3 特权级的栈。

每个任务的每个特权级下只能有一个栈，不存在一个任务的某个特权级下存在多个同特权级栈的情况。也就是说，一共 4 个特权级，一个任务"最多"有 4 个栈。既然一个 TSS 代表一个任务，每个任务又有 4 个栈，那为什么 TSS 中只有 3 个栈：ss0 和 esp0、ss1 和 esp1、ss2 和 esp2？它们分别代表 0 级栈的段选择子和偏移量、1 级栈的段选择子和偏移量、2 级栈的段选择子和偏移量。大家看，我在前面说的一个任务最多拥有 4 个栈，并不是所有的任务都是这样的。

要想搞清楚这个问题，得先弄明白 TSS 中记录的 3 个栈是用来干吗的。

刚才已经说过，特权级在变换时，需要用到不同特权级下的栈，当处理器进入不同的特权级时，它自动在 TSS 中找同特权级的栈，你懂的，TSS 是处理器硬件原生的系统级数据结构，处理器当然知道 TSS 中哪些字段是目标栈的选择子及偏移量。

特权级转移分为两类，一类是由中断门、调用门等手段实现低特权级转向高特权级，另一类则相反，是由调用返回指令从高特权级返回到低特权级，这是唯一一种能让处理器降低特权级的情况。

对于第 1 种——特权级由低到高的情况，由于不知道目标特权级对应的栈地址在哪里，所以要提前把目标栈的地址记录在某个地方，当处理器向高特权级转移时再从中取出来加载到 SS 和 ESP 中以更新栈，这个保存的地方就是 TSS。处理器会自动地从 TSS 中找到对应的高特权级栈地址，这一点对开发人员是透明的，咱们只需要在 TSS 中记录好高特权级的栈地址便可。

也就是说，除了调用返回外，处理器只能由低特权级向高特权级转移，TSS 中所记录的栈是转移后的高特权级目标栈，所以它一定比当前使用的栈特权级要高，只用于向更高特权级转移时提供相应特权的栈地址。进一步说，TSS 中不需要记录 3 特权级的栈，因为 3 特权级是最低的，没有更低的特权级会向它转移。

不是每个任务都有 4 个栈，一个任务可有拥有的栈的数量取决于当前特权级是否还有进一步提高的可能，即取决于它最低的特权级别。比如 3 特权级的程序，它是最低的特权级，还能提升三级，所以可额外拥有 2、1、0 特权级栈，用于将特权分别转移到 2、1、0 级时使用。2 特权级的程序，它还可以提升两级，所以可额外拥有 1、0 特权级栈，用于将特权级分别转移到 1、0 级时使用。以此类推，1 特权级的程序，它可以额外拥有 0 特权级栈，0 特权级已经是至高无上了，只有这一个 0 级栈。以上所说的低特权级转向高特权级的过程称为"向内层转移"，想想 4 个特权级划分的同心圆就知道了，高特权级位于里面。

对于第 2 种——由高特权返回到低特权级的情况，处理器是不需要在 TSS 中去寻找低特权级目标栈的。其中一个原因我想您已经猜到了：TSS 中只记录 2、1、0 特权级的栈，假如是从 2 特权级返回到 3 特权级，上哪找 3 特权级的栈？另一方面的原因是低特权级栈的地址其实已经存在了，这是由处理器的向高特权级转移指令（如 int、call 等）实现的机制决定的，换句话说，处理器知道去哪里找低特权级的目标栈，等我把后面内容"啰嗦完"您就知道了。

由于特权级向低转移后，处理器特权级有了变化，同样也需要将当前栈更新为低特权级的栈，它如何找到对应的低特权级栈呢？正常情况下，特权级由低向高转移在先，由高向低返回在后，即只有先向更高特权级转移，才能谈得上再从高特权级回到低特权级，否则没有"去"就谈不上"回"（宁可被骂啰嗦，我也要说清楚）。当处理器由低向高特权级转移时，它自动地把当时低特权级的栈地址（SS 和 ESP）压入了转移后的高特权级所在的栈中（随着以后深入学习大家会明白这一点），所以，当用返回指令如 retf 或 iret 从高特权级向低特权级返回时，处理器可以从当前使用的高特权级的栈中获取低特权级的栈段选择子及偏移量。由高特权级返回低特权级的过程称为"向外层转移"。

当下次处理器再进入到高特权级时，它依然会在 TSS 中寻找对应的高特权级栈，而 TSS 中栈指针值都是固定的，每次进入高特权级都会重复使用它们。也就是说，即使曾经转移到高特权级下用过高特权级栈，处理器也不会自动把该高特权级栈指针更新到 TSS 中，因为在从高特权级返回时，处理器需要把栈更新为低特权

级的栈选择子及 esp 指针，而原先在段寄存器 SS 和寄存器 esp 中高特权级下的栈段选择子及指针会被处理器自动丢弃。换句话说，如果想保留上一次高特权级的栈指针，咱们得自己手动更新 TSS 中相应栈的数据。

上面光说处理器从 TSS 中找更高特权级的栈地址，那处理器是怎样找到 TSS 的？

TSS 是硬件支持的系统数据结构，它和 GDT 等一样，由软件填写其内容，由硬件使用。GDT 也要加载到寄存器 GDTR 中才能被处理器找到，TSS 也是一样，它是由 TR（Task Register）寄存器加载的，每次处理器执行不同任务时，将 TR 寄存器加载不同任务的 TSS 就成了。至于怎么加载以及相关工作原理，目前咱们用不到，还是放在后面说比较合适。

您看，正是由于处理器提供了硬件方面的框架，所以很多工作都是“自动”完成的，虽然操作系统看上去是底层的技术，但其实也属于“应用型”开发。

好啦，TSS 中有关特权级的内容就说到这，为了不干扰大家学习特权级，TSS 的其他方面将会在后续章节中逐步说明。

5.4.3　CPL 和 DPL 入门

我们在工作中，公司都给员工配有员工卡，此员工卡就是员工身份的“标签”，用它来出入公司、食堂就餐等。给公司创造价值的是员工的生产力，不是员工卡，员工卡只是公司人事部门管理员工的一种手段而已。

现在说计算机，既然是用特权级来维护计算机世界的和平，那总该给每个被管理的对象加个特权“标签”，也就是说 CPU 得知道谁的特权高，谁的特权低，这样才能辨识出是否有低特权级的程序越级访问高特权级资源的违法行为。

员工的标签体现在工卡，计算机特权级的标签体现在 DPL、CPL 和 RPL，下面咱们围绕这几个概念展开讨论。

先看看访问者的特权标签在哪里。

最初我们刚接触保护模式的时候，最先感受到的区别是访问内存不像在实模式下那么自由、直接啦，在保护模式下有了段描述符，访问内存得先经过它才行，它的作用是通过各种属性描述一段内存区域，该描述符就相当该内存区域的“身份证”。

前情提要，x86 访问内存的机制是“段基址：偏移地址”，无论是实模式，还是保护模式，都要遵循此方式。在实模式下，段基址直接写在段寄存器中，而在保护模式下，段寄存器中的不再是段基址，而是段选择子，通过该选择子从 GDT 或 LDT 中找到相应的段描述符，从该描述符中获取段的起始地址。大伙儿还记得选择子的结构吧，第 0～1 位是 RPL 字段，第 2 位是 TI 位，第 3～15 位是段描述符索引，好啦，回忆到此为止。咱们要关注的就是 RPL 字段，它就是请求特权级，后面讲 RPL 的时候咱们会细说。请求特权级中的“请求”是个动词，只有具备“能动性”的访问者才能做出动作。

话说回来了，谁是访问者？计算机中，具备“能动性”的只有计算机指令，只有指令才具备访问、请求其他资源的能力，指令便是资源的请求者。指令“请求”、“访问”其他资源的能力等级便称之为请求特权级，指令存放在代码段中，所以，就用代码段寄存器 CS 中选择子的 RPL 位表示代码请求别人资源能力的等级。代码段寄存器 CS 和指令指针寄存器 EIP 中指向的指令便是当前在处理器中正在运行的代码，所以，位于 CS 寄存器中选择子低 2 位的值不仅称为请求特权级，又称为处理器的当前特权级，也就是说处理器的当前特权级是 CS.RPL。

处理器的当前特权级的真实面目是什么？

在 CPU 中运行的是指令，其运行过程中的指令总会属于某个代码段，该代码段的特权级，也就是代码段描述符中的 DPL，便是当前 CPU 所处的特权级，这个特权级称为当前特权级，即 CPL（Current Privilege Level），它表示处理器正在执行的代码的特权级别。除一致性代码段外（后面会说），转移后的目标代码据段的 DPL 是将来处理器的当前特权级 CPL。

指令最终是用处理器执行的，执行到不同特权的代码，处理器的特权级就换到不同的等级。所以，当前特权级实际上是指处理器当前所处的特权级，是指处理器的特权角色，更形象一点地说，是指 CPU 当前在计算机世界中的特权地位。再次提醒大伙儿，在任意时刻，当前特权级 CPL 保存在 CS 选择子中的 RPL 部分。

当前特权级存储在 CS.RPL 中，谁为代码段寄存器 CS 赋值的呢？要回答上面的问题，得先搞清楚处理器的当前特权级为什么会变化。

当前正在运行的代码所在的代码段的特权级 DPL 就是处理器的当前特权级，当处理器从一个特权级的代码段转移到另一个特权级的代码段上执行时，由于两个代码段的特权级不一样，处理器当前的特权身份起了变化，这就是当前特权级 CPL 改变的原因。好像说得有点神秘，其实就是使用了那些能够改变程序执行流的指令，如 int、call 等，这样就使 CS 和 EIP 的值改变，从而使处理器执行到了不同特权级的代码。不过，特权转移可不是随便进行的，处理器要检查特权变换的条件，这里咱们暂不讨论条件是什么，因为它需要过会儿结合 RPL 一块说，等小弟把 RPL 给大伙儿说清楚了再解释这个“条件”不迟。当处理器特权级检查的条件通过后，新代码段的 DPL 就变成了处理器的 CPL，也就是目标代码段描述符的 DPL 将保存在代码段寄存器 CS 中的 RPL 位。

总之，代码是资源的请求者，代码段寄存器 CS 所指向的是处理器中当前运行的指令，所以代码段寄存器 CS 中选择子的 RPL 位称为当前特权级 CPL，这是再合理不过的事了。

RPL 变成了 CPL，似乎有点晕是吗？其实只是代码段寄存器 CS 中的 RPL 是 CPL，其他段寄存器中选择子的 RPL 与 CPL 无关，因为 CPL 是针对具有“能动性”的访问者（执行者）来说的，代码是执行者，它表示访问的请求者，所以 CPL 只存放在代码段寄存器 CS 中低 2 位的 RPL 中。

以上几段内容想表达的就是处理器的当前特权级 CPL 为什么放在 CS.RPL 中，CPL 和 CS.RPL 有什么关系，不信您再回头看一遍。

大多数情况下，处理器都是在“访问者”访问“受访者”时进行特权检查，访问者（某个代码段）的特权就是当前特权级 CPL，在进行特权检查时，都会以 CPL 为基础，说到这里，不禁我要自问一下，既然 CPL 就是目标代码段的 DPL，那处理器总该有个初始 CPL 吧？答案是肯定的，特权级是保护模式下的概念，处理器进入保护模式后才有的当前特权级，让我们回忆下处理器是怎么进入保护模式的，也许答案自然就揭晓啦。

在这之前，我和大家说过，MBR 从 BIOS 接过接力棒的时候，它已经处于 0 特权级啦，其实这么说不太合适，毕竟在保护模式下才有特权级，而 MBR 运行在实模式下，还谈不上特权级。BIOS 将控制权交给 MBR，是用一个远跳转指令实现的，即 jmp 0：0x7c00，也就是说进入 MBR 后，段寄存器 CS 会被替换为 0，如果此时把 CS 看作是选择子的话，RPL 的值为 0，也就是说“相当于”处于 0 特权级，段寄存器 CS 为 0 的情况一直持续到在 loader 中进入保护模式之后的第一条指令：流水线刷新跳转。该跳转语句如图 5-48 所示。

```
19     SELECTOR_CODE equ (0x0001<<3) + TI_GDT + RPL0
略...
132     jmp dword SELECTOR_CODE:p_mode_start
```

▲图 5-48　loader.S 中流水线刷新跳转语句

大家看图中第 132 行的 jmp 指令，段选择子为 SELECTOR_CODE，其 RPL 的值为 RPL0，RPL0 定义在 include/boot.inc 中，其值为 0。选择子的索引部分值为 1，表示对应 GDT 中第 1 个段描述符，该描述符的 DPL 为 0（它是用 include/boot.inc 中的 DESC_DPL_0 定义的，图中未展示）。

在跳转之前，CS 为 0，其低 2 位 RPL 部分为 0，也就是 CPL 为 0，当执行跳转指令 jmp dword SELECTOR_CODE: p_mode_start 时，目标代码段，即第 1 个段描述符的 DPL 为 0，与当前特权级一致（处理器会根据 CPL、RPL、DPL 做特权级检查，此检查过程咱们在介绍完 RPL 时再讨论），处理器允许转移，所以新的特权级依然是 0，该值保存在段寄存器 CS 的低 2 位，这就是特级级转移的粗略过程，也是进入保护模式后特权为 0 的来龙去脉。

说完了 CPL，咱们再看看，受访者的特权标签在哪里。

在段描述符中有一个属性还为该内存标明了特权等级，这就是段描述符中的 DPL 字段的作用，它就是受访者的特权标签。话说，不仅段描述符中有 DPL 字段，以后所介绍的所有描述符都有 DPL。

DPL，即 Descriptor Privilege Level，描述符特权级，这下您清楚为什么 DPL 字段在段描述符中占 2 位的原因了吧，两位能表示 4 个组合，00b、01b、10b、11b，所有特权级都齐了。

计算机是人发明的，用人的思想来理解计算机原理是再合适不过的。拿校园生活举例，班长权限比班主任低，班长有权限安排学生打扫卫生，班主任有权限查看学生成绩。班长没权限查看学生成绩，但班主任有权限安排学生打扫卫生。这就是拥有高特权级的事物可以访问同级或更低特权级资源，而低特权级的

事物无法访问高特权级资源的典型例子。

在计算机中也一样，DPL 是段描述符所代表的内存区域的“门槛”权限，访问者能否迈过此门槛访问到本描述符所代表的资源，其特权级至少要等于这个门槛，访问者特权能否大于该门槛？这要看受访资源是代码，还是数据啦。不难想像，只有具备“能动”行为的访问者才具备访问的能力，在计算机中真正的访问者是硬件 CPU，而指挥 CPU 行为（访问谁及如何访问）的是具有可执行能力的指令代码，数据是不能访问别人的，所以我们再强调一下访问者就是代码段中的指令，这对理解当前特权级非常重要。

访问者任何时候都不允许访问比自己特权更高的资源，无论受访资源是数据，还是代码。在不涉及 RPL 的前提下，下面咱们要分情况讨论啦。

对于受访者为数据段（段描述符中 type 字段中未有 X 可执行属性）来说：

只有访问者的权限大于等于该 DPL 表示的最低权限才能够继续访问，否则连这个门槛都迈不过去。比如，DPL 为 1 的段描述符，只有特权级为 0、1 的访问者才有资格访问它所代表的资源，特权为 2、3 的访问者会被 CPU 拒之门外。

对于受访者为代码段（段描述符中 type 字段中含有 X 可执行属性）来说：

只有访问者的权限等于该 DPL 表示的最低权限才能够继续访问，即只能平级访问。任何权限大于或小于它的访问者都将被 CPU 拒之门外。这是为什么呢？自问自答之前先明确一个概念，对于受访者为代码段一这说法，实际上是指处理器从当前运行的代码段上转移到受访者这个目标代码段上去执行，并不是说把该目标代码段当数据一样访问，在真实物理机器上，代码段通常情况下是不被当成数据来处理的，但确实可以这么做（话说虚拟机中会把代码当成数据来处理）。

咱们先说为什么比它特权级更高的代码也无法“访问”它，即转移到它上面运行。

代码指令代表 CPU 的行为，低特权级的代码能做的事，高特权级代码也能做，换句话说高特权的代码不需要低特权代码的帮助，正常情况下 CPU 没有理由先自降等级后再去做某事。代码段是 CPU 执行的指令，不是数据，这里所说的“受访者为代码段”其实就是指 CPU 从访问者所在的段转移到该代码段上去执行。举个例子，CPU 若相当于汽车，代码则相当于司机，它指挥 CPU 前进的方向。段的变换相当于换了司机，特权级较高的代码段相当于技术水平较高的 F1 车手，特权级较低的代码段相当于技术水平较低的普通司机，这辆车为了充分展示性能，活得更加精彩，它始终希望它的搭档是驾驶技术高超的 F1 车手，要是把搭档降级为普通司机，它可万万不能答应啊。

不过，凡事都有例外的时候，这是唯一一种处理器会从高特权降到低特权运行的情况：处理器从中断处理程序中返回到用户态的时候。

中断处理都是在 0 特权级下进行的，因为中断的发生多半是外部硬件发生了某种状况或发生了某种不可抗力事件而必须要通知 CPU 导致的，所以，在中断的处理过程中需要具备访问硬件的能力，在大多数情况下只有 CPU 处于 0 特权级才能访问硬件，这是因为 eflags 寄存器中的 IOPL 位的值通常被设置为 0（该位的作用就是限制访问 IO 端口的最低特权级），并且 TSS 中不存在 IO 位图，有关这部分后面马上会讲到。再者，有些中断处理中需要的指令只能在 0 特权级下使用，这部分指令称为特权指令，所以中断发生后其处理的过程必须在 0 特权级下进行。用户进程是在 3 特权级，在运行用户程序时若发生了中断，CPU 会暂停用户程序的执行，随后 CPU 就会自动由 3 特权级进入到 0 特权级，在 0 特权级下将执行用户程序时的现场环境（也就是著名的概念：上下文）保存起来（这个保存上下文的动作可以由 CPU 通过 TSS 完成，这是 CPU 在硬件上提供的功能，但其效率并不高，所以大多数操作系统都是自己写代码手动保存上下文环境），待中断处理完成后，CPU 会恢复用户程序的执行，也就是说会回到 3 特权级。以后在讲了中断和用户进程时大伙儿会更清楚这一点。

现在大家知道了，除了从中断处理过程返回外，任何时候 CPU 都不允许从高特权级转移到低特权级。再结合之前咱们所说的大前提，访问者任何时候都不允许访问比自己特权更高的资源，代码段也是资源，只不过可以让 CPU 转移过去执行而已，所以，比目标代码段特权级低的访问者也会被拒绝访问目标代码段。综上所述，对于受访问者为代码段的情况，只能是平级访问。也就是说，假如当前特权级为 2，只能转移到 DPL 为 2 特权级的代码段上运行，转移到 0、1、3 特权级都会被处理器拒绝。

不过本质上来说，代码能否运行与代码本身的特权等级并无关系，不同等级下的代码段中的机器码都

是一样的，特权级只是写在描述符的 DPL 中，并没有写在机器码中，所以高特权级的代码并不比低特权级的代码显得“高大上”，目标段（代码段或数据段）的特权级仅仅是在被访问时由处理器检查一次，之后再无用途，所以特权级并不影响处理器执行指令。把计算机资源划分成不同等级，只是让处理器知道自己正在处理的资源的“份量”有多重，必须用同样的身份来执行，不能儿戏。

如果处理器仅能平移代码段的话，另外三个特权级的代码将没有机会运行啦。如何穿过特权屏障呢？处理器又提供了多种方式用于从低特权的代码转移到高特权代码。

处理器的特权级升高之后，程序想干什么就干什么，多少都觉得有点恐怖，有没有一种好办法，既执行高特权级代码段上的指令，又不提升特权级？一种方式是利用一致性代码段。

什么是一致性代码段？早在当初介绍段描述符结构时就已经提过它了，不过确实只是提了一下而已，以至于您可能完全没有印象。在段描述符中，如果该段为非系统段（段描述符的 S 字段为 0），可以用 type 字段中的 C 位来表示该段是否为一致性代码段。C 为 1 时则表示该段是一致性代码段，C 为 0 时则表示该段为非一致性代码段。上面所提到的代码段是非一致性代码段，所以只能平级转移。

一致性代码段也称为依从代码段，Conforming，用来实现从低特权级的代码向高特权级的代码转移。一致性代码段是指如果自己是转移后的目标段，自己的特权级（DPL）一定要大于等于转移前的 CPL，即数值上 CPL≥DPL，也就是一致性代码段的 DPL 是权限的上限，任何在此权限之下的特权级都可以转到此代码段上执行。这似乎很奇怪，但却在意料之中，奇怪的是在低特权级访问高特权级居然是可以的，而意料之中的是这才能实现了代码转移。该关系用公式表示如下：

在数值上，CPL≥一致性代码段的 DPL

一致性代码段的一大特点是转移后的特权级不与自己的特权级（DPL）为主，而是与转移前的低特权级一致，听从、依从转移前的低特权级，这就是它称为“依从、一致”的原因。也就是说，处理器遇到目标段为一致性代码段时，并不会将 CPL 用该目标段的 DPL 替换。

大家注意啦，既然是转移到特权级更高的一致性代码段后 CPL 不变，这说明这种转移本身并没有提升特权级，只是可以跑到特权级更高的代码段中去执行指令，对计算机而言并未因特权级升高而产生潜在危险，所以在特权级检查过程中，请求者的 RPL 并不参与。

特权级检查发生在访问者访问受访者的一瞬间，只检查一次，在检查过后，在该段上以后的执行过程中也不会再被检查，处理器也并不会因为执行了高特权级的代码而觉得自己了不起，在它眼里任何级别的指令都是一样的。特权级就是一个个的关卡，仅仅是进门时的检测而已，与关卡后面的代码和数据无关。这种情况类似必须要刷工卡才能进公司一样，即使不刷工卡，您的工位还是那个工位，电脑还是那个电脑，您照样还是可以使用它们，这些资源并不会有什么改变，它们与咱们是否刷工卡是无关的。所以，尽管转移到一致性代码段后 CPL 还保持为原来的低特权级并未提升，但也没什么好奇怪的，毕竟目的是运行目标代码段上的指令，达到目的就行了，何必在意身份呢。

顺便说一句，代码段可以有一致性和非一致性之分，但所有的数据段总是非一致的，即数据段不允许被比本数据段特权级更低的代码段访问。

按理说，把代码划分为不同等级就是为了等级变换，如果还以卑微的身份去运行高特权级的代码，似乎不够体面，人家孙悟空为了面子还号称为齐天大圣呢，其实 CPU 中实现特权级变换的“门路”多着呢，还有 4 个“门”。

5.4.4 门、调用门与 RPL 序

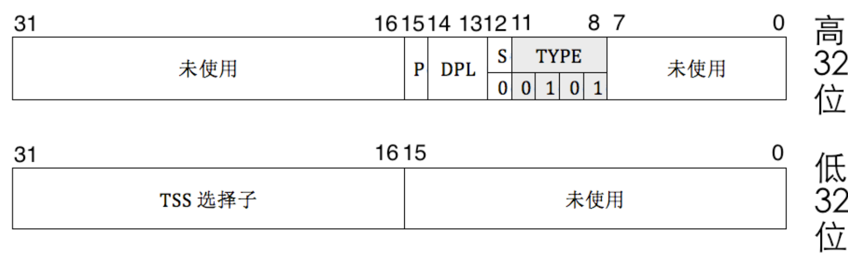
多次想把调用门和 RPL 分开单独说，但几次尝试都没有成功，我发现它们之间是紧耦合、密不可分的，RPL 的产生主要是为了解决系统调用时的“越权”问题，系统调用的实现方式中，以调用门和中断门最为适合。由于以后我们将用中断门实现自己的系统调用，故在此本着扩充知识面的目的给大伙儿介绍调用门，通过调用门的实例，让大伙儿理解特权级那点事儿。

处理器只有通过“门结构”才能由低特权级转移到高特权级，处理器就是这样设计的，我们必须要遵循它的用法，对处理器来说，操作系统只是它的应用而已。

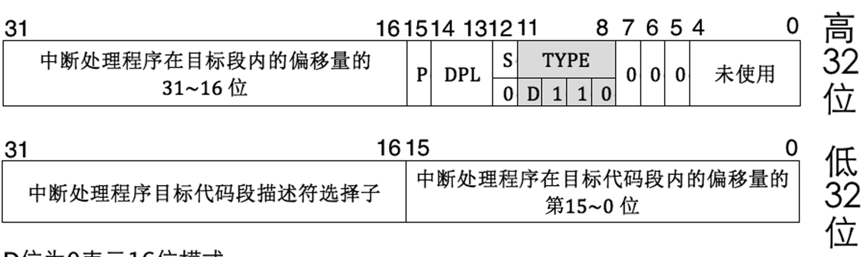
门结构是什么呢？就是记录一段程序起始地址的描述符。

描述符有多种，刚才所说的一致性代码段，虽然它里面全是代码，但它本身是内存段，并不是指具体的一段例程，所以可以用“段描述符”来“描述”。还有一种称为“门描述符”的结构，用来描述一段程序。进入这种神奇的“门”，处理器便能转移到更高的特权级上。

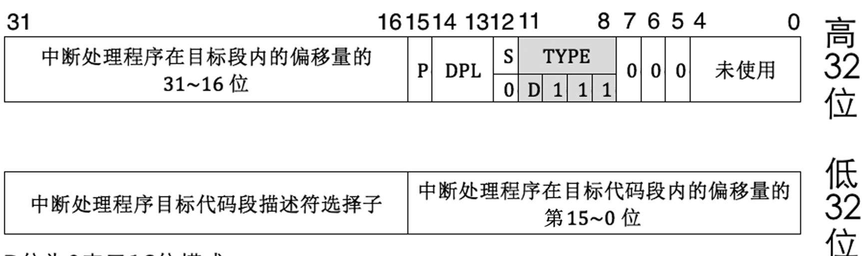
门描述符同段描述符类似，都是 8 字节大小的数据结构，用来描述门中通向的代码。一共有 4 种门结构，图 5-49～图 5-57 这 4 张图所示是 4 种门描述符的结构，咱们先看图，然后再简要介绍。



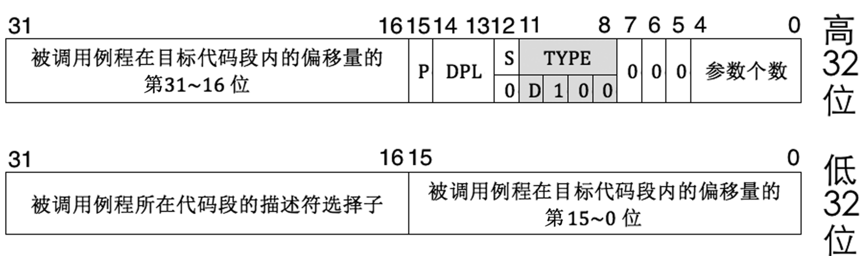
▲图 5-49　任务门描述符



▲图 5-50　中断门描述符



▲图 5-51　陷阱门描述符



▲图 5-52　调用门描述符

大家看图 5-52 中的 4 种门描述符，它们与段描述符最大的不同是除了任务门外，其他三种门都是对应到一段例程，即对应一段函数，而不是像段描述符对应的是一片内存区域。任何程序都属于某个内存段，所以程序确切的地址必须用“代码段选择子+段内偏移量”来描述，可见，门描述符基于段描述符，例程是用段描述符来给出基址的，所以门描述符中要给出代码段的选择子，但光给出基址远远不够，还必须给出例程的偏移量，这就是门描述符中记录的是选择子和偏移量的原因。

任务门描述符可以放在 GDT、LDT 和 IDT（中断描述符表，后面章节在介绍中断时大伙儿就清楚了）中，调用门可以位于 GDT、LDT 中，中断门和陷阱门仅位于 IDT 中。

任务门、调用门都可以用 call 和 jmp 指令直接调用，原因是这两个门描述符都位于描述符表中，要么是 GDT，要么是 LDT，访问它们同普通的段描述符是一样的，也必须要通过选择子，因此只要在 call 或 jmp 指令后接任务门或调用门的选择子便可调用它们了。陷阱门和中断门只存在于 IDT 中，因此不能主动调用，只能由中断信号来触发调用。

任务门有点特殊，它用任务 TSS 的描述符选择子来描述一个任务，有关 TSS 的内容会在用户进程部分介绍。除任务门之外，另外的三个门描述符都是用代码段选择子及偏移地址来描述一段程序例程的。但是，无论是哪种门描述符，它们中所记录的信息都已经可以确定所描述的对象（例程或任务）了，所以在被调用时，CPU 都会忽略调用指令中的偏移量。例如：假设某调用门描述符位于 GDT 中第 1 个位置，这样的指令“call 0x0008：0x1234”，在调用此调用门时，偏移量 0x1234 会被 CPU 忽略。

提供了 4 种门的原因是它们都有各自的应用环境，但它们都用来实现从低特权级的代码段转向高特权级的代码段，咱们这里也只讨论有关特权级的功用。

1．调用门

call 和 jmp 指令后接调用门选择子为参数，以调用函数例程的形式实现从低特权向高特权转移，可用来实现系统调用。call 指令使用调用门可以实现向高特权代码转移，jmp 指令使用调用门只能实现向平级代码转移。

2．中断门

以 int 指令主动发中断的形式实现从低特权向高特权转移，Linux 系统调用便用此中断门实现，以后咱们在实现中断时会展开细说。

3．陷阱门

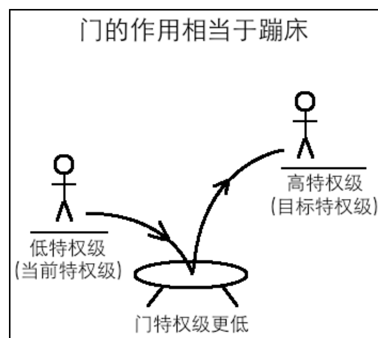
以 int3 指令主动发中断的形式实现从低特权向高特权转移，这一般是编译器在调试时用，本书中咱们不用过多关注。

4．任务门

任务以任务状态段 TSS 为单位，用来实现任务切换，它可以借助中断或指令发起。当中断发生时，如果对应的中断向量号是任务门，则会发起任务切换。也可以像调用门那样，用 call 或 jmp 指令后接任务门的选择子或任务 TSS 的选择子。

坦白说，现代操作系统很少用到调用门和任务门，在咱们的系统中也只用到了中断门，而陷阱门是供调试器用的，咱们并未打算支持应用程序的调试，一方面工作量较大，另一方面违背咱们的初衷，就是想通过更少的代码了解操作系统原理。

不知道大伙儿有没有想过，为什么可以使用门结构进入高特权级呢？这肯定是 CPU 硬件电路中写好的规则，具体我也不清楚，也不需要搞清楚，但我是这样理解的，举个例子，用门提升特权级，就像站在高处的台子上往蹦床上跳一样，人会被蹦床弹得比台子还高。关键点：台子的高度位于蹦床和目标高度之间，至少得和蹦床一样高，这样人才能被弹得更高。其效果如图 5-53 所示。



▲图 5-53　门的作用

实际上，门也是按照这个蹦床原理实现的，我们把“门”分成门槛和门框来比喻。

门的“门槛”是访问者特权级的下限，访问者的特权级再低也不能比门描述符的特权级 DPL 低，否则访问者连门都进不去，更谈不上使用调用门。门描述符的 DPL 特权

级要低于或等于当前特权级 CPL，即数值上 CPL≤门的 DPL，此处可见，门描述符相当于数据段描述符一样，只允许比自己特权级高或相同特权级的程序访问。

门的“门框”是访问者特权级的上限，访问者的特权级再高也不能比门描述符中目标程序所在代码段的 DPL 高，否则本身的特权级就比目标代码特权级还高的话，还使用门干吗？而且真要是这样的话，这意味是特权级由高向低转移了，这绝对是被禁止的。也就是说，门中包含的目标程序所在的段的特权级 DPL 要高于或等于当前特权级 CPL，即数值上 CPL≥目标代码段 DPL，进门之后，处理器将以目标代码段 DPL 为当前特权级 CPL，因此进门之后，就能“一步登天”啦。

如上所述，并不是任何当前特权级都可以使用门结构的，在使用门结构之前，处理器要例行公事做特权级检查，参与检查的不只是 CPL 和 DPL，还有 RPL，为了说清楚这个检查过程，咱们得先介绍下 RPL。

RPL，即请求特权级，为了解释清楚，咱们得多花点工夫好好说道说道。

我们本节始终在说特权级转移，处理器从一个特权级转移到另一个特权级，任意时刻处理器所处的特权级称为当前特权级。重复叙述的目的是强调当前特权级是对处理器而言的概念，并不是对代码段而言。当前特权级 CPL 是指处理器任意时刻的身份地位，其变化的原因是处理器从某一特权级的代码段转移到另一特权级的代码段上运行，代码段的特权级 DPL 是未来处理器的 CPL。

各种门结构存在的目的就是为了让处理器提升特权级，这样处理器才能够做一些低特权级下无法完成的工作。比如，当用户程序想读取硬盘文件时，由于处理器在执行用户程序时所处的特权级为 3，一般情况下操作系统不允许用户程序操作硬盘。此时必须由用户代码指挥处理器使用某种门结构（如调用门）进入 0 特权级，在提升了处理器的 CPL 之后才能控制硬盘，读取文件。是不是说得有些抽象？其实就是用户程序进行系统调用使处理器进入内核态执行内核服务。

当处理器提升为 0 特权级时，任何事情都能做，是最强大，同时也是最危险的状态，如果用户程序通过某种门结构使处理器进入到 0 特权级，它很有可能会被 3 特权级的用户程序利用，这样用户程序就有机会访问 0 特权级下的数据。

调用门是一个描述符，称为门描述符，其中记录的是内核服务程序所在代码段的选择子及在代码段中的偏移地址。门描述符定义在全局描述符表 GDT 和局部描述符表 LDT 中，所以，要想使用调用门，就要通过门描述符的选择子，这一点和访问数据段类似，总之，保护模式下离不开描述符，有描述符就离不开选择子。

我们平时很少有人直接和调用门打交道，大多数程序员甚至都不知道调用门是怎么回事，所以在接触调用门时通常会感到有些吃力，这是由三方面造成的。

（1）我们大多数情况下是用的高级语言编程，编译器或集成开发环境为我们做了太多的工作，大大方便了我们的编码方式，而调用门是在汇编语言下使用的方法，不做底层开发的话我们根本就碰不到它。

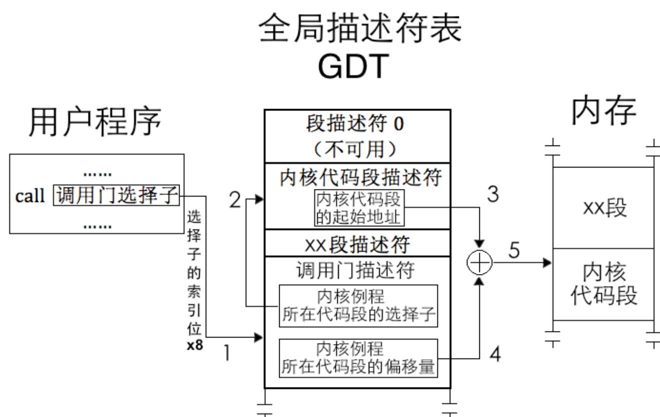
（2）调用门是用来实现系统调用的，但为了兼容等原因，我们平时接触的操作系统很少使用调用门实现系统调用，如 Linux 就是用中断门代替。我们顶多听说过中断门，对调用门了解少之又少。

（3）调用门一般在过去多段模型下使用，大多数情况下需要为调用门指定段选择子。而现在操作系统为了方便，早已经采用了平坦模型，所有用户进程共享几个选择子，比如用户代码段选择子和用户数据段选择子各一个，由所有用户进程共享，用户进程不需要再提供选择子，所以调用门可以用中断门代替了。

综上所述，调用门是在汇编语言中使用的，能发挥其特长的场所是多段模型，若没有此方面的编程经验，大家先提前有个印象，也没什么复杂的，仅仅是大家很少接触而已。

调用门描述符是位于 GDT 或 LDT 中的，所以要调用它就要通过门描述符选择子才行。通常，调用门由 call 指令或 jmp 指令后接门描述符选择子来调用，后面还会说到，咱们先初识一下调用门吧。

操作系统可以利用调用门实现一些系统功能（但现代操作系统用调用门实现系统调用并不是主流，一般是用中断门实现系统调用），用户程序需要系统服务时可以调用该调用门以获得内核帮助。结合图 5-52 所示调用门描述符结构，调用门描述符记录的就是一段函数例程地址，以函数例程地址所在的代码段选择子及所在代码段的偏移量组成，只是该例程的提供者是处于 0 特权级下的内核。调用门的执行流程如图 5-54 所示。



▲图 5-54　调用门执行流程

给大伙儿说下调用门内部执行流程，如图 5-54 所示，从调用门选择子到获取到内核代码的地址，共经历了 5 个步骤。在用户程序中有一句代码“call 调用门选择子”，call 指令可以使用调用门，参数就是调用门的选择子，该选择子指向 GDT 或 LDT 中的某个门描述符，不管选择子中的 TI 位是 0，还是 1，我们暂且认为它是指向 GDT 中的调用门。处理器用门描述符选择子的高 13 位（索引位）乘以 8 作为该描述符在 GDT 中的偏移量，再加上寄存器 GDTR 中的 GDT 基地址，最终找到了门描述符的地址，它位于 GDT 中从 0 起的第 3 个描述符位置，该位置如图 5-54 所示是门描述符。在该描述符中记录的是内核例程的地址。我们知道，在保护模式下描述某个内存地址是离不开选择子和偏移量的，所以，门描述符中记录的是内核例程所在代码段的选择子及偏移量。处理器再用代码段选择子，重复之前的步骤，用选择子中高 13 位的索引值乘以 8，再加上 GDT 基址，所得到的地址为该代码段选择子所指向的内核代码段描述符地址，在该内核代码段描述符中找到内核代码段基址，用它加上门描述符中记录的内核例程在代码段中的偏移量，最终得到内核例程的起始地址。

以上为讨论方便，执行过程未涉及分页机制。

既然门描述符用来指向某个内核例程，是例程就需要参数，我们讨论下怎样在用户的 3 特权级下为 0 特权级下的内核程序传递参数。

不同特权级下处理器用不同的栈，处理器处于 3 特权级下时要用 3 特权级下的栈，处理器处于 0 特权级下要用 0 特权级下的栈，这么做的原因主要是怕不同特权下的数据混在同一个栈中交叉引用会比较混乱，而且所有特权级共用一个栈的话很容易使栈溢出。参数最初是由用户程序以压栈的形式提交给调用门的，也就是说参数是保存在用户栈中的，目前读者先这么理解就行，在以后讨论调用约定时大家就明白了。陷入内核后，由于特权级变化，导致使用的栈也要跟着变化，不能再使用用户程序下的 3 特权级栈了，处理器要用 0 特权级下的栈，所以，处于 0 特权级下的内核程序需要在其对应的 0 级栈中获取参数。大家要清楚，用户传入参数是在 3 特权级下做的，参数在 3 特权级栈中，内核服务程序是在 0 特权级下，它需要在 0 特权级栈中获取参数，3 特权级的用户程序怎样能越权将参数压入 0 特权级下的栈呢？这种访问相当于 CPL 为 3 的进程访问 DPL 为 0 的数据段，数值上 CPL>DPL，处理器会引发异常的。有没有读者这样想，让操作系统再提供一个调用门专门负责传入参数到 0 级栈……先不说这样是否可行，不能否认这的确有点荒唐，我们的初衷是想通过调用门执行内核程序，现在却为了使用“调用门”而去调用另外一个调用门，这显然很不科学。处理器的设计者也看到了这一点，为了方便软件开发人员，处理器在固件上实现参数的自动复制，即，将用户进程压在 3 特权级栈中的参数自动复制到 0 特权级栈中。

所以，您看到在图 5-52 中，其高 32 位的起始处有个“参数个数”，这是处理器将用户提供的参数复制给内核时需要用到的，参数在栈中的顺序是挨着的，所以处理器只需要知道复制几个参数就行了，这就是调用门描述符中“参数个数”的作用，它是专门给处理器准备的。该位是用 5 个 BIT 来表示的，所以最多可传递 31 个参数。

调用门可以用 call 指令和 jmp 指令调用，jmp 属于一去不回头的指令，基本上用在不需要从调用门返回的场合。call 指令由于会在栈中留下返回地址，所以在执行 retf 指令时还能返回。

调用门的使用形式是："call 或 jmp 指令+调用门描述符选择子"，由于门描述符中已经记录了程序的偏移地址，所以即使在 call 和 jmp 指令后面接偏移地址也会被处理器忽略，如 call selector_ gate:offset,offset 会被忽略。

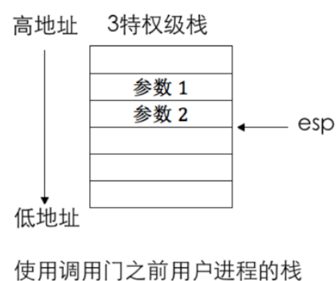
5.4.5　调用门的过程保护

调用门涉及两个特权级，先是转移前的低特权级，这是程序调用"调用门"时的 CPL，再就是转移后的目标特权级，这是由门描述符中选择子对应的目标代码段的 DPL 决定的。通过调用门有可能是平级转移，这是有可能的，并不是说调用门的使命就是向更高特权级转移，内核程序也可以调用"调用门"，内核是 0 特权级，特权没法再高了。所以，调用门未必都向高特权级转移，也可能是 CPL 为 0，目标代码段的 DPL 也为 0，这样就是平级转移。顺便说一句，即使是通过调用门也不能由高特权级转向低特权级，特权级检查时就无法通过，只有通过返回指令如 retf 或 iret 才能够做到由高特权级转移到低特权级，调用门只能向平级或更高级转移。

接下来，咱们看看在用户进程中通过 call 指令调用"调用门"的完整过程，以下我们只讨论 32 位模式。

假设用户进程要调用某个调用门，该门描述符中参数的个数是 2，也就是用户进程需要为该调用门提供 2 个参数才行。调用前的当前特权级为 3，调用后的新特权级为 0，所以调用门转移前用的是 3 特权级栈，调用后用的是 0 特权级栈。

（1）现在为此调用门提供 2 个参数，这是在使用调用门前完成的，目前是在 3 特权级，所以要在特权级栈中压入参数，分别是参数 1 和参数 2，如图 5-55 所示。



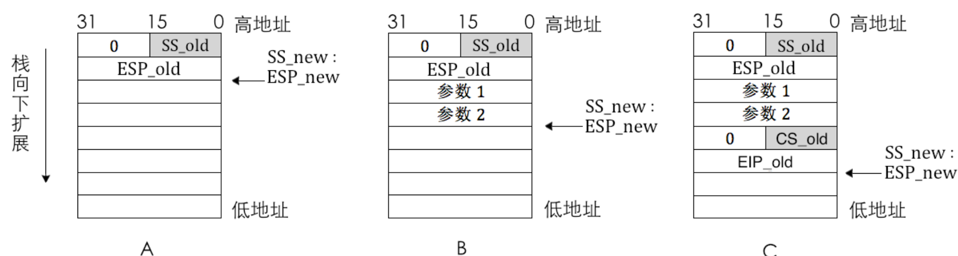
▲图 5-55　调用门控制转移前用户栈

（2）在这一步骤中要确定新特权级使用的栈，新特权级就是未来的 CPL，它就是转移后的目标代码段的 DPL。所以，根据门描述符中选择子对应的目标代码段的 DPL，如前假设，这里 DPL 为 0，处理器自动在 TSS 中找到合适的栈段选择子 SS 和栈指针 ESP，它们作为转移后新的栈。为方便叙述，将它们记作 SS_new、ESP_new。

（3）检查新栈段选择子对应的描述符的 DPL 和 TYPE，如果未通过检查则处理器引发异常。

（4）如果转移后的目标代码段 DPL 比 CPL 要高，说明栈段选择子 SS_new 是特权级更高的栈，这说明需要特权级转换，需要切换到新栈，为叙述方便，将旧栈段选择子记作 SS_old，旧栈指针记作 ESP_old。由于转移前的旧栈段选择子 SS_old 及指针 ESP_old 得保存到新栈中，这样在高特权级的目标程序执行完成后才能通过 retf 指令恢复旧栈。但此时新栈的选择子 SS_new 和 ESP_new 还未加载到栈段寄存器 SS 和栈指针 esp 中，只能在使用新栈后才能将 SS_old 和 ESP_old 保存到新栈中，所以处理器先找个地方临时保存 SS_old 和 ESP_old，之后将 SS_new 加载到栈段寄存器 SS，esp_new 加载到栈指针寄存器 esp，这样便启用了新栈。

（5）在使用新栈后，将上一步中临时保存的 SS_old 和 ESP_old 压入到当前新栈中，也就是 0 特权级栈。由于咱们讨论的是 32 位模式，故栈操作数也是 32 位，SS_old 只是 16 位数据，将其高 16 位用 0 填充后入栈保存。如图 5-56 A 所示。



▲图 5-56　特权级转移新栈

（6）在这一步中要将用户栈中的参数复制到转移后的新栈中，根据调用门描述符中的“参数个数”决定复制几个参数。参数复制后的效果如图 5-56 B 所示。

（7）由于调用门描述符中记录的是目标程序所在代码段的选择子及偏移地址，这意味着代码段寄存器 CS 要用该选择子重新加载，处理器不管新的选择子和任何段寄存器（包括 CS）中当前的选择子是否一致，也不管这两个选择子是否指向相同的段，只要段寄存器被加载，段描述符缓冲寄存器就会被刷新，从而相当于切换到了新段上运行，这是段间远转移，所以需要将当前代码段 CS 和 EIP 都备份在栈中，这两个值分别记作 CS_old 和 EIP_old，由于 CS_old 只是 16 位数据，在 32 位模式下栈操作数大小是 32 位，故将其高 16 位用 0 填充后再入栈。这两个值是将来恢复用户进程的关键，也就是从内核进程中返回时用的地址。效果如图 5-56C 所示。

（8）一切就绪，只差运行调用门中指向的程序啦，于是，把门描述符中的代码段选择子装载到代码段寄存器 CS，把偏移量装载到指令指针寄存器 EIP。

至此，处理器终于从用户程序转移到了内核程序上，实现了特权级由 3 到 0 的转移，开始执行门描述符中对应的内核服务程序。

话说如果在第 2 步中处理器发现是平级转移，比如内核程序调用“调用门”的情况，这是 0 级到 0 级的平级转移，处理器并不会更新当前栈，也就是说不会从 TSS 中再次选择同级的栈载入 SS 和 ESP，处理器只是把此转移当成直接远转移，于是跨过中间几步，直接做第 7 步，将 CS 和 EIP 压入当前栈中。

当处理器执行完内核服务程序时，接下来咱们再看看处理器是如何从高特权级返回到低特权级的。

由 call 指令调用的调用门，已经在高特权级栈中留下了返回地址及低特权级的栈，所以可以用 retf 指令将返回地址从栈中弹出到 CS 和 EIP，将低特权级栈地址弹出到 SS 和 ESP 中。

不过，话又说回来了，处理器是被所有任务共享的，它在不同任务之间来回切换，它根本不知道自己刚刚执行的是哪个任务，更不知道自己是刚刚从低特权级通过调用门过来的。那它靠什么保证正确的执行流程呢？答案是，靠人，卖了那么大的关子，结果答案竟然如此出人意料。其实，只有人才知道程序的流程，所以咱们在工作中才能有条不紊地按照流程编码。处理器只知道执行你交给它的指令，至于合不合理，那是由人来控制的。就拿这个调用门来说，如果调用的时候用 call 指令，这说明还要从目标程序中返回，也就是说调用门对应的程序，其结尾应该是 retf 指令才合理。如果该调用门是用 jmp 调用的，它用在一去不回的场合，对应的调用门中就不需要用 retf 指令。在这个例子中，是用 call 指令调用的，所以程序能回到正确的执行流上，是 retf 指令起的作用。

想想看，如果程序真是从调用门过来的，由于在经过调用门的时候已经做了特权级检查，按理说程序返回到低特权级时，就不需要再做特权检查了。可是经过上面的分析，大家已经了解了处理器并不知道自己刚刚做了什么，它不清楚自己是刚才经过调用门进到高特权级的，所以它也不知道应该原路返回到低特权级。你懂的，程序之所以能正确回到低特权级，是因为咱们人为地加了 retf 指令来保证的。所以，处理器有权怀疑自己到底是不是从调用门过来的，为以防万一，它从调用门回来时也要做特权级检查。

下面是利用 retf 指令从调用门返回的过程。

（1）当处理器执行到 retf 指令时，它知道这是远返回，所以需要从栈中返回旧栈的地址及返回到低特权级的程序中。这时候它要进行特权级检查。先检查栈中 CS 选择子，根据其 RPL 位，即未来的 CPL，判断在返回过程中是否要改变特权级。

（2）此时栈顶应该指向栈中的 EIP_old。在此步骤中获取栈中 CS_old 和 EIP_old，根据该 CS_old 选择子对应的代码段的 DPL 及选择子中的 RPL 做特权级检查，规则不再赘述。如果检查通过，先从栈中弹出 32 位数据，即 EIP_old 到寄存器 EIP，然后再弹出 32 位数据 CS_old，此时要临时处理一下，由于所有的段寄存器都是 16 位的，当然包括 CS，所以丢弃 CS_old 的高 16 位，将低 16 位加载到 CS 寄存器。此时栈指针 ESP_new 指向最后一个参数。

（3）如果返回指令 retf 后面有参数，则增加栈指针 ESP_new 的值，以跳过栈中参数，按理说，“retf+参数”是为了跳过从低特权级栈中复制到当前高级栈中的参数，如参数 1 和参数 2，所以 retf 后面的参数应该等于参数个数*参数大小。此时，栈指针 ESP_new 便指向 ESP_old。

（4）如果在第 1 步中判断出需要改变特权级，从栈中弹出 32 位数据 ESP_old 到寄存器 ESP。同样寄

存器 SS 也是 16 位的，故再弹出 32 位的 SS_old，只将其低 16 位加载到寄存器 SS，此时恢复了旧栈。相当于丢弃寄存器 SS 和 ESP 中原有的 SS_new 和 ESP_new。

至此，程序流程便返回到了之前调用“调用门”的低特权级程序。

注意，如果在返回时需要改变特权级，将会检查数据段寄存器 DS、ES、FS 和 GS 的内容，如果在它们之中，某个寄存器中选择子所指向的数据段描述符的 DPL 权限比返回后的 CPL（CS.RPL）高，即数值上返回后的 CPL>数据段描述符的 DPL，处理器将把数值 0 填充到相应的段寄存器。

为什么在这种情况下处理器会在寄存器中填充 0 呢？

原因是这样的，处理器的特权级检查只发生在往段寄存器中加载选择子的时候，检查通过之后，再从该段中进行后续数据访问时就不需要再进行特权检查了。这是比较合理的，哪有访问一次数据就检查一次特权级的，这样太低效了，这和现实中过安检是一样的，只要能通过“安全门”，之后便认为此人不再是危险分子了，总之检查只发生在关卡处。处理器是按照人的思维设计的，所以只在往段寄存器加载选择子时检查一次就够了，之后对该段的读写访问将不再受限。非常合理是吧？但问题也出在这。

当用户程序通过调用门进入内核程序时，处理器的 CPL 变为 0，内核程序为了返回用户进程所需要的数据，内核必然会访问自己的数据段，其 DPL 为 0，所以在数据段寄存器中的选择子必须是内核自己的选择子，其 RPL 为 0。这样 CPL = RPL =0，数值上小于等于 DPL，这样特权级检查才会成功。

比如在内核服务例程中这样访问内核自己的数据段：

```
mov ax,selector_DS_RPL0
mov ds,ax       ;更新数据段寄存器 DS，使其指向内核数据段
                ;（此时会进行特权级检查）
mov ax, [0x1234];此时地址 0x1234 的默认段寄存器是指向内核数据段
```

在这之后，数据段寄存器 ds 一直指向内核自己的数据段。

当处理器执行 retf 指令从内核态返回时，处理器顶多是把栈中低特权级的 CS 选择子、EIP 的值、SS 选择子和 ESP 的值分别重新加载到寄存器 CS、EIP、SS 及 ESP。像 DS 等数据段寄存器是不会被更新的。特权级检查是发生在往数据段寄存器中加载选择子的时候，此时 DS 中的选择子依然是指向内核数据段的，只要不重新加载段寄存器 DS，特权级检查就不会发生，所以返回到用户态后，此时虽然 CPL=3，但用户进程依然可以直接访问内核数据段中的数据，这还得了。

一个可行的解决办法是操作系统代码在使用任何一个数据段寄存器时，先将其入栈，然后再更新其选择子，在使用完毕之后，操作系统再将压入栈中的选择子出栈恢复到该段寄存器。比如：

```
push ds;
mov ax,selector_DS_RPL0
mov ds,ax
…
开始访问内核数据段
执行内核服务代码
…
在内核服务例程执行完成后
pop ds     ;恢复 ds
retf
```

但这只是用软件来避免此问题的方法，谁也不能确保软件都会这样做，甚至包括操作系统。（题外话，在 Linux 中就不用调用门，只用中断门来实现系统调用，在中断处理程序中会手动更新所有寄存器，也就是手动任务恢复上下文，所以不会存在此类问题。）

处理器不相信第三方的软件都会处理好此问题，甚至操作系统也不值得相信，毕竟操作系统也是软件，处理器只相信它自己。

于是，在返回时若需要改变特权级，处理器必须要检查所有数据段中的选择子，若 DS、ES、FS 和 GS 中选择子所指向的数据段描述符的 DPL 权限比目标特权级高，处理器将把数值 0 填充到相应的段寄存器。

把段寄存器填充为 0 就解决问题了吗？显然不是，目的是为了让处理器引发异常。

其实这和全局描述符表 GDT 有关，听兄弟慢慢道来。

GDT 中第 0 个段描述符是不可用的，称之为哑描述符，为什么第 0 个不可用呢？其实原因在很久以

前就说过了，这是为了避免选择子忘记初始化而误选了第 0 个段描述符，出问题时通常很难排查出来。比如，如果选择子忘记初始化的话，即选择子的高 13 位段描述符索引位为 0，这表示第 0 个段描述符，选择子中的 TI 为 0，这表示在 GDT 中检索，RPL 虽然也为 0，但咱们不关心，这种 0 值的选择子就会在 GDT 中检索到第 0 个哑描述符，从而引发处理器的异常。

以上在段寄存器中填充 0，便是利用选择子为 0 引发异常的原理。

5.4.6 RPL 的前世今生

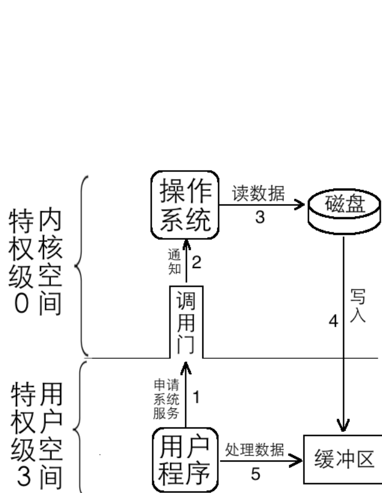
由于咱们用不到调用门，所以大伙儿对它的了解不用太深，我们以调用门举例，是为了引出为什么要用请求特权级 RPL。

调用门大致原理就是这样，咱们举个例子来说明潜在的危险。看看仅仅靠 CPL 和 DPL 行不行。

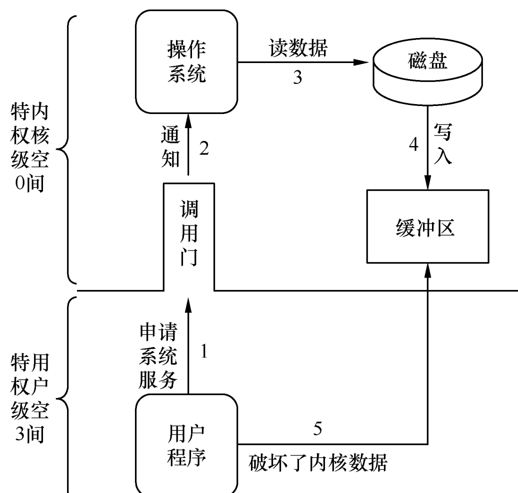
调用门 A 可以帮助用户程序把硬盘某个扇区的数据写入到用户指定的内存缓冲区中，说得似乎很“底层”的样子，其实就是读文件的功能。因此，此调用门需要三个参数，分别是磁盘逻辑扇区号 LBA、内存缓冲区所在数据段的选择子、内存缓冲区的偏移地址。正常情况下，其流程如图 5-57 所示。

图中的步骤看上去好和谐，似乎哪个环节都不会出问题。当 3 特权级的用户程序使用调用门后，处理器进入了 0 特权级，也就是 CPL 为 0，这已经是最高的特权级了，不管受访者的 DPL 是多少，如果特权检查仅仅靠 CPL 和 DPL 这两项的话，数值上 CPL≤DPL，这时候处理器可以说已经是傲视天下了，可以访问并获得任何资源。何况内存缓冲区是 3 特权级的用户数据段，这更不在话下，处理器是有资格将数据写入缓冲区的。

其实只是正常情况下是这样，因为正常情况下用户所提交的缓冲区的选择子指向用户自己的数据段。说到这估计您也想到了，问题来了，倘若用户程序怀着一颗有非法企图的心，将参数——缓冲区所在数据段的选择子，用内核数据段的选择子代替（用多个选择子测试几次数据便可推测内核数据段），这样就把内核破坏了。结果如图 5-58 所示，缓冲区位于内核数据区啦。



▲图 5-57 调用门执行过程



▲图 5-58 用户进程非法访问

再举个例子，这个例子可能不现实，但这是可能的，仅仅是为了暴露出问题。

假如用个内存复制功能的调用门，就可以轻易地获得内核数据，让用户进程获得内核数据，这也是万万不可的，比如某个调用门可以完成类似 memcpy 的功能，参数是：数据源所在段的选择子及数据源的偏移地址，目的地址所在段的选择子及目的地址的偏移地址和拷贝的字节数据量，一共 5 个参数。

当用户进程通过调用门陷入内核后，处理器的特权级变为 0 级，还是那句话，如果特权检查仅仅靠 CPL 和 DPL 这两项的话，这时候处理器可以访问并获得任何资源。

正常情况下，用户进程若老老实实地提交自己的数据段选择子作为数据源，那当然是一位遵纪守法的好市民。但如果提交的数据源参数是内核数据段选择子，用户必然会获取到内核的数据，这等于内核完全暴露。

您看，仅仅靠 CPL 和 DPL 还真不行，毫无安全可言，咱们分析下问题出在哪里。

咱们看看目前的两个必须保证的客观条件。

（1）用户不能访问系统资源，不能越俎代庖去做操作系统的事，这是安全的底线，操作系统必须保证用户程序不会乱动系统资源。看来，用户程序要想做点“大事”，必须要由操作系统出面啦。

（2）处理器必须要陷入内核才能帮助用户程序做“大事”，所以处理器的当前特权级会变成至高无上的 0 特级级。

以上是两个雷打不动的客观条件，必须要这样才行，当处理器在 0 特级下时，跟谁要数据谁都得乖乖地交出来，也就是说在现有的客观条件下，看上去无计可施啦，似乎得增加条件才行。

仔细想想，出现问题的原因是：受访者不知道访问者的真实身份，在受访者看来，是 0 特权级的操作系统想要数据，它还以为请求者是操作系统呢。实际情况是请求者为 3 特权级下的用户程序，内核程序只是代替用户程序来拿数据的。即资源的真正请求者是用户程序，而它却是破坏者。用户程序所处的特权级为 3，按道理低特权级的访问者是不被允许访问这些高特权级的内核资源，问题就出在这，我们要想办法让受访问者知道，真正请求资源的是谁，它到底有没有资格获取这些数据。如果它有权限的话就把资源给它，否则直接拒绝请求。

RPL 完美地解决了这个问题。之前只是简单地说了下 RPL，现在该是揭开它的本来面目的时候了。

RPL 是谁？RPL，Request Privilege Level，请求特权级，这么说有点歧义，其实它代表真正请求者的特权级，我的言外之意是说 RPL 其实是代表真正资源需求者的 CPL，大伙儿继续听我说。

以后在请求某特权级为 DPL 级别的资源时，参与特权检查的不只是 CPL，还要加上 RPL，CPL 和 RPL 的特权必须同时大于等于受访者的特权 DPL，即：

数值上 CPL≥DPL 并且 RPL≤DPL

有没有读者想过，RPL 不是位于选择子中的低 2 位吗？用户提供的选择子，难道用户不能伪造个 RPL？

大家想想看，GDT、LDT 等全局系统数据表都是由操作系统构建的，只有操作系统知道表中段描述符、门描述符的索引，理所当然选择子也应该由操作系统构建，由操作系统分派给用户进程使用就行了。所以，一般情况下选择子都是由操作系统提供的，也就是说控制权在操作系统手里。当用户程序请求操作系统服务，如果需要提交选择子作为参数，为安全起见，操作系统会把选择子中的 RPL 改为用户程序的 CPL，为此，处理器还提供了修改 rpl 的相关指令，后面会介绍。

有没有同学问，什么时候用户需要自己提交选择子？其实这还是比较久远的事了，在非平坦模型下编程时，用户进程的数据和代码单独存在于某些代码段和数据段中，肯定是要用选择子来指定的。如果用户进程不涉及提交选择子做参数的话，操作系统服务程序内部所用的选择子肯定都是操作系统自己的构造的，操作系统对自己的东西还是不用质疑的。比如说在平坦模型下，整个 4GB 内存是一个段，操作系统为所有用户进程构建了两个用户级的 RPL 为 3 的选择子，分别指向 4GB 的用户数据段和 4GB 的用户代码段。因为用户程序在自己的虚拟地址空间中运行，各用户进程的虚拟地址不冲突，所以各用户程序共用这两个选择子就够了，也就是说用户进程在申请系统服务时无需提供选择子，从而，操作系统在系统服务程序内部就用这两个用户级的选择子便可以搞定一切。

上面这段内容算是小扩展了一下，咱们还是回来继续说最基本的东西。

用户程序的 CPL 是不会骗人的，不可能伪造，它起始是由操作系统在加载用户程序时赋予的，记录在段寄存器 CS 中的低 2 位，就是 RPL 的位置，而 CS 寄存器只能通过 call、jmp、ret、int、sysenter 等指令修改，即使改的话，用户程序也只能在 3 级特权下折腾，只要用户进程不请求操作系统服务，它的 CPL 是不会变的，当它申请了系统服务，如果提交了选择子作为参数，选择子中的 RPL 也会被操作系统修改为用户进程的 CPL。所以，即使用户程序提交了个伪造的选择子也没用，其 RPL 会被操作系统用其 CPL 替换，还其“真身”。

刚才提到了修改 RPL 的指令，在汇编中就是 arpl 指令，此指令用来修改选择子中的 RPL。虽然在咱们系统中用不上，还是给介绍下比较踏实，其用法是：

arpl 通用寄存器/16 位内存，16 位通用寄存器

目的操作数可以是任意一个通用寄存器或 16 位大小的内存，它们用来存储用户提交的选择子，源操

作数是 16 位通用寄存器，里面存储用户进程的代码段寄存器 CS 的值。实际此指令操作数就变成了：

arpl 用户提交的选择子，用户段寄存器 CS 的值

这样一来，不管用户是否伪造了选择子中的 RPL，RPL 都通通被置为正确的值。

还有一点要说的是这里的源操作数，也就是用户 CS 寄存器的值是从栈中获取到的（肯定不是用户自己主动提交的，它可没有那么“老实”）。现在要剧透一下啦，是这样的：用户程序为 3 特权级，它当请求系统服务时，需要陷入内核，处理器会进入 0 特权级，此时，由于是远转移，处理器会自动将段寄存器 CS 和 EIP 的值压入栈中，而且特权级也发生了变化，所以一块压入的还有寄存器 SS、ESP，这 4 项都是用户程序的寄存器环境，这方面内容在讲完这部分后我们马上会说。这样操作系统便可以在栈中获取到用户程序的 CS 寄存器的值，以便将其作为 arpl 指令的源操作数。

另外，除了加载用户程序时，在其他时段的 CPL 是由目标代码段的 DPL 变成的，即切换到新特权代码段后，新代码段的 DPL 被存储到段寄存器 CS 中的低 2 位，就是 RPL 的位置。其实这再合理不过了，CPU 是切换到新的特权级代码段上运行了，身份变了，当然要用新代码段的 DPL 做 CPL。

RPL 引入的目的是避免低特权级的程序访问高特权级的资源。有了 RPL 之后，我们说一下现在的特权检查的步骤，看看是否能解决之前的问题。

DPL 相当于权限的门槛，它代表进入本描述符所对应内存区域的最低权限，任何想迈过这个门槛的人，它的 RPL 和 CPL 权限必须都要大于等于 DPL，即数值上 CPL≤DPL && RPL≤DPL。

特权级检查实际上就是让 CPU 检查数值上 CPL≤DPL && RPL≤DPL 是否成立，这是工程师给处理器设置的规则，用来检查当前请求者和真正的资源需求方是否都具有访问受访者的资格。这么做的原因是当前请求者和资源需求方有可能不是同一个人，也许只是某人拜托了一位有能力的人作为自己的代理人去获取资源，代理人肯定是无所不能的，哪块数据都能得到，但受访者不知道请求者是否是代理，引入 RPL 的目的是让受访者知道，不管当前请求者是不是代理，即使当前请求者是替别人来拿数据的，那个人也必须得有获得数据的权限才行，否则照样驳回当前的请求者。

特权级检查发生在什么时候呢？如何被触发？

32 位保护模式下对内存的访问要通过段描述符，段描述符中有 DPL，这是内存的关卡，咱们现实生活中的检查也是在关卡处执行的，所以处理器的特权检查，都是只发生在往段寄存器中加载选择子访问描述符的那一瞬间，所以，RPL 放在选择子中是多么的合理。这里所说的加载选择子，是指任何访问，无论是代码，还是数据。处理器的特权检查只发生在访问前的一瞬间，这和现实生活中是一样的，通过检查之后再也不管了，直到遇到新的关卡，否则执行一步指令就要检查一次特权级，处理器啥活都甭干了。

总结下不通过调用门、直接访问一般数据和代码时的特权检查规则，对于受访者为代码段时：

- 如果目标为非一致性代码段，要求：

数值上 CPL=RPL=目标代码段 DPL

- 如果目标为一致性代码段，要求：

数值上（CPL≥目标代码段 DPL && RPL≥目标代码段 DPL）

受访者若为代码，只有在特权级转移时才会被用到，所以有关代码的特权检查都发生在能够改变代码段寄存器 CS 和指令指针寄存器 EIP 的指令中，即这些指令要么改变 EIP，要么改变 CS 和 EIP。例如 call、jmp、int、ret、sysexit 等能改变程序执行流的指令。

对于受访者为数据段时：

数值上（CPL ≤目标数据段 DPL && RPL ≤ 目标数据段 DPL）

栈段的特权级检查比较特殊，因为在各个特权级下，处理器都要有相应的栈（后面会说到），也就是说栈的特权等级要和 CPL 相同。所以往段寄存器 SS 中赋予数据段选择子时，处理器要求 CPL 等于栈段选择子对应的数据段的 DPL，即数值上 CPL = RPL = 用作栈的目标数据段 DPL。

受访者若为数据，特权级检查会发生在往数据段寄存器中加载段选择子的时候，数据段寄存器包括 DS 和附加段寄存器 ES、FS、GS。

举个例子，mov ds，ax 时便会触发特权级检查。ax 中的值被当作选择子，处理器会拿 ax 中的低 2 位，

即 RPL 和 CPL 分别与 ax 中选择子所指向的段描述符的 DPL 做比较，如果满足 RPL≤DPL && CPL≤DPL，选择子才能被加载到 DS 中。

大家不要把 CPL 和 RPL 搞混了，不要误以为都是对同一个程序而言的，它们也许不都属于同一个程序。RPL 是位于选择子中的，所以，要看当前运行的程序在访问数据或代码时用的是谁提供的选择子，如果用的是自己提供的选择子，那肯定 CPL 和 RPL 都出自同一个程序，如果选择子是别人提供的，那就有可能 RPL 和 CPL 出自两段程序。CPL 是对当前正在运行的程序而言的，而 RPL 有可能是正在运行的程序，也可能不是。在一般情况下，如果低特权级不向高特权级程序提供自己特权级下的选择子，也就是不涉及向高特权级程序“委托、代理”办事的话，CPL 和 RPL 都来自同一程序。但凡涉及“委托、代理”，进入 0 特权级后，CPL 是指代理人，即内核，RPL 则有可能是委托者，即用户程序，也有可能是内核自己。还是拿之前说过的调用门 A 举例，某用户进程运行在 3 特权级，它想通过调用门读取硬盘上某个文件到它自己的数据缓冲区中。它需要向该调用门提供 3 个参数：文件所在的硬盘扇区号、用于存储文件的缓冲区所在的数据段选择子以及缓冲区的偏移地址。用户进程只能把与自己同一特权的数据段作为缓冲区，所以该缓冲区所在段的 DPL 为 3，其选择子的 RPL 必然为 3。进入调用门后，处理器的 CPL 由运行用户进程时的 3 级变成内核态的 0 级，当内核从硬盘上读取完数据后，需要将其写入用户的缓冲区中。缓冲区的选择子是由用户提供的，其 RPL 如上所述为 3，缓冲区所在段的 DPL 为 3，此时 CPL 为 0，即数值上（CPL≤DPL && RPL≤DPL）成立，于是写入成功。大伙儿看到了，RPL 是用户进程提供的，而往缓冲区写数据时 CPL 指的是内核，不是同一个程序。

也许您对上述例子中使用调用门时的特权级检查感兴趣，后面会有详细的过程，咱们先举几个简单的特权级检查的例子，循序渐进。

对于一般不通过使用门，较“直接”的访问：

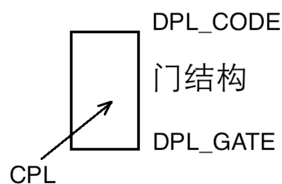
比如，当前运行的是用户进程，也就是处理器的 CPL（CS.RPL）为 3，用户进程自己的数据段 DPL 为 3，此时进程想往自己的数据段中写入数据，进程就要提供自己数据段的选择子到段寄存器 DS（选择子通常是由操作系统提供的），由于此时的写入只是同级操作，用户自己便能够完成，不需要系统功能调用，所以操作系统自然也不知道此事。CPL=RPL=3，DPL=3，故数值上满足（RPL≤DPL && CPL≤DPL），写入没问题。

还是这个用户进程，现在它想往内核数据段搞点破坏，想写入数据，所以它就要提供内核数据段的选择子。按理说内核数据段选择子是不会暴露给用户的，但用户能猜出来，所以可以伪造一个内核数据段的选择子。因为 GDT 大小是有限的，除第 0 个描述符不能用以外，其他的都可以挨个试，哈哈，也许有更好的方法。用户进程在 3 级环境下想直接写入内核数据段，它伪造的选择子的 RPL 为 0，故 CPL =3，RPL= 0，内核数据段 DPL=0，故数值上不满足 RPL≤DPL && CPL ≤DPL，从而写入失败，处理器抛出 GP 异常，即一般保护性错误。其实只要 CPL 为 3，伪造的 RPL 是多少都不行，CPL 才是短板。

在举例调用门的例子之前，咱们得把之前遗留的工作完成。前面咱们在介绍调用门的特权检查时暂停了一下，因为涉及到 RPL。现在跟大家说道说道。

调用门本身是个描述符，所以它有个 DPL，我们暂且将其记作 DPL_GATE。而且门用来指向一段程序，程序所在的代码段本身也有个 DPL，暂且将其记作 DPL_CODE。所以，你懂的，使用调用门会涉及对这两个 DPL 的特权级检查。

首先，门的作用相当于蹦床，门槛低，但能跳到更高的特权级之上。DPL_GATE 相当于特权级的门槛，表示特权下限，DPL_CODE 相当于特权级的门框，表示特权上限，如图 5-59 所示。



▲图 5-59　CPL 与门特权级关系

如图 5-59 所示，对于门来说，处理器在检查特权级时：

（1）要求 CPL 和 RPL 在 DPL_GATE 和 DPL_CODE 之间。

（2）RPL 只用在进调用门时和 DPL_GATE 比较一次，不参与和 DPL_CODE 比较。原因一方面是门描述符选择子只用来索引门结构，如调用门，不是用来索引任何内存段的，过了这个调用门的检查，此选择子再也不需要了，不会有潜在的威胁。另一方面，门描述符选择子中的 RPL 即使是被用户进程伪造的，为了成功，也只能改成比 CPL 更高的特权，否则没意义，连门槛

DPL_GATE 都进不了。所以，即使 RPL 是伪造的，CPL 也为相对较低的特权，它相当于特权短板。在接下来的检查中，如果要求 RPL 和 CPL 同时比门框特权级 DPL_CODE 低的话，为了避免 RPL 是伪造的，仅用 CPL 和 DPL_CODE 比较就够了，无需 RPL 参与。

如上所述，处理器对于门的特权检查，公式为：

（1）数值上 DPL_GATE≥CPL≥DPL_CODE（对应上面的 1）。

并且

（2）数值上 RPL≤DPL_GATE（对应上面的 2）。

以上两点要同时满足。

注意，门描述符中的选择子只是用来指向目标程序所在的代码段的，而真正请求者是使用调用门的程序，所以门描述符中选择子的 RPL 并不参与特权级检查，因为它并不代表真正请求者。

调用门是用 call 指令和 jmp 指令来调用的，jmp 指令只能用于平级跳转，call 可以跨越特权级。对于一致性代码，由于转移过去后 CPL 并不会有变化，等同于同级转移，所以特权及检查对于 call 和 jmp 来说是没区别的，都是用上面的两个公式。可是对于非一致性代码就不同了，转移到非一致性代码段后，会涉及到特权级变化，call 指令支持跨越特权的转移，其特权检查还是上面那两个公式，而 jmp 只适用于特权级平级转移，CPL 必须等于目标代码段的 DPL。所以，对于 jmp 的特权级检查，公式如下：

（1）数值上 DPL_GATE≥CPL = DPL_CODE。

并且

（2）数值上 RPL≤DPL_GATE。

有关特权级的部分咱们就说完了，也许有同学还不是很清楚 CPL、RPL、DPL 的关系，下面咱们通过调用门的例子把这一过程梳理下。

假设当前处理器正在 DPL 为 3 的代码段上运行，即正在运行用户程序，故处理器当前特权级 CPL 为 3。此时用户进程想获取安装的物理内存大小，该数据存储在操作系统的数据段中，该段 DPL 为 0。由于当前运行的是用户程序，CPL 为 3，所以无法访问 DPL 为 0 的数据段。于是它使用调用门向系统救助。调用门是操作系统安装在全局描述符表 GDT 中的，为了让用户进程可以使用此调用门，操作系统将该调用门描述符的 DPL 设为 3。该调用门只需要一个参数，就是用户程序用于存储系统内存容量的缓冲区所在数据段的选择子和偏移地址。调用门描述符中记录的就是内核服务程序所在代码段的选择子及在代码段内的偏移量。用户进程用“call 调用门选择子”的方式使用调用门，此调用门选择子是由操作系统提供的，该选择子的 RPL 为 3，此时如果用户伪造一个调用门选择子也没用，因为此选择子是用来索引门描述符的，并不用来指向缓冲区的选择子，调用门选择子中的高 13 位索引值必须要指向门描述符在 GDT 中的位置，选择子中低 2 位的 RPL 伪造也没意义，因为此时 CPL 为 3，是短板，以它为主。此时处理器便进行特权级检查，CPL 为 3，RPL 为 3，门描述符 DPL 为 3，即数值上（CPL≤DPL && RPL≤DPL）成立，初步检查通过。接下来还要再将 CPL 与门描述符中选择子所对应的代码段描述符 DPL 比较，这是调用门对应的内核服务程序的 DPL，为叙述方便将其记作 DPL_CODE。由于 DPL_CODE 是内核程序的特权级，所以 DPL_CODE 为 0，CPL 为 3，即数值上满足 CPL≥DPL_CODE，CPL 比目标特权级低，检查通过，该用户程序可以用调用门，于是处理器的当前特权级 CPL 的值用 DPL_CODE 代替，记录在 CS.RPL 中，此时 CPL 变为 0。接下来，处理器便以 0 特权级的身份开始执行该内核服务程序，由于该服务程序的参数是用户提交的缓冲区所在的数据段的选择子及偏移量，为避免用户将缓冲区指向了内核的数据区，安全起见，在该内核服务程序中，操作系统将这个用户所提交的选择子的 RPL 变更为用户进程的 CPL，也就是指向缓冲区所在段的选择子的 RPL 变成了 3。前面说过，参数都是内核在 0 级栈中获得的，虽然用户进程将缓冲区的选择子及偏移量压在了 3 特权级栈中，但由于调用门的特权级变换，参数已经由处理器在固件一级上自动复制到 0 特权级栈中了。用户的代码段寄存器 CS 也在特权级发生变化时，由处理器自动压入到 0 特权级栈中，所以操作系统需要的参数都可以在自己的 0 特权级栈中找到。用户缓冲区的选择子修改过后，接下来内核服务程序将用户所需要的内存容量大小写到这个选择子和用户提交的偏移量对应的缓冲区。如果用户程序想搞破坏，所提交的这个缓冲区选择子指向的目标段不是用户进程自己的数据段，而是内核数

据段或内核代码段，由于目标段的 DPL 为 0，虽然此时已在内核中执行，CPL 为 0，但选择子 RPL 已经被改为 3，数值上不满足 CPL≤DPL && RPL≤DPL，往缓冲区中的写入被拒绝，处理器引发异常。如果用户程序提交的缓冲区选择子确实指向用户程序自己的数据段，DPL 则为 3，数值上满足 CPL≤DPL && RPL≤DPL，往缓冲区中的写入则会成功。如果中断服务程序内部再有访问内核自己内存段的操作，还会按照数值上（CPL≤DPL && RPL≤DPL）的策略进行新一轮的特权检测。

通常，如果不是用户程序向内核提交缓冲区地址来接收数据的话，内核不会主动访问用户的内存段，多是访问自己的数据段或代码段，内核服务程序中若访问内核自己的内存段，由于内存段的 DPL 为 0，所以段选择子的 RPL 也必须为 0。

在介绍远调用门的执行流程后，似乎也该结束了。不过，不知道大伙儿有没有疑问，为什么 jmp 指令即使通过调用门也只是特权平级转移？

jmp 有自己的远转移形式，即可以跨段跳转，但它的用途是一去不回头的那种场合，它不像 call 指令那样还能回来。call 指令还能回来的原因是在栈中压入了返回地址，将来在用 ret 系列的指令时，可以从栈中弹出返回地址到 EIP 或 CS 和 EIP 中，这取决于用的是否是跨段返回指令 retf。

特权级转移只能由低转向高，当进入高特权级时，栈也要切换到高特权下的栈，假如 jmp 可以转移到高特权级的代码段，处理器也跟着切换到了高特权级的栈，想象一下会怎样呢？之前说过，除可以用 retf 指令从高特权返回到低特权外，处理器是不允许从高特权级向低特权级转移的，但 jmp 指令不会在高特权级的栈中留下返回地址，它只管去，不管回来，所以即使用 retf 指令也找不到回来的路，无法回到低特权级下。这等于低特权下的资源再也无法访问了。

也许这就是它的宿命，它就是被设计用于平级转移的，所以 jmp 只用在不需要特权级变化，且不从调用门返回的场合。

CPL、DPL、RPL 都说完了，不知道大家有没有理解，其实我当初学习特权级的时候对于 RPL 还是有些模糊的。为了帮助大伙儿理解，我在半夜失眠时特意想了一个例子。

不知道大伙儿学车了没有，报考驾校也要有个年龄限制，即使考 C 本 B 本也要分年龄的。假如某个小学生 A（用户进程）特别喜欢开车，他就是想考个驾照，可驾校的门卫（调用门）一看他年龄太小都不让他进门，连填写报名登记表的机会都没有，怎么办？于是他就求他的长辈 B（内核）帮他去报名，长辈的年龄肯定够了，门卫对他放行，他来到驾校招生办公室后，对招生人员说要帮别人报名。人家招生人员对 B 说，好吧，帮别人代报名需要出示对方的身份证（RPL），于是长辈 B 就把小学生 A 的身份证（现在小孩子就可以申请身份证，只是年龄越小有效期越短，因为小孩子长得快嘛）拿出来了，招生人员一看，年纪这么小啊，不到法制学车年纪呢，拒绝接收。这时候驾校招生人员的安全意识开始泛滥了，以纵容小孩子危险驾驶为名把长辈 B 批评了一顿（引发异常）。

5.4.7　IO 特权级

在保护模式下，处理器中的“阶级”不仅体现在数据和代码的访问，还体现在指令中。

一方面将指令分级的原因是有些指令的执行对计算机有着严重的影响，它们只有在 0 特权级下被执行，因此被称为特权指令（Privilege Instruction）。比如 hlt 指令，它可以让计算机停机，处理器只信任操作系统，所以它不得不放在 0 特权级下。同类的指令还有 lgdt、lidt、ltr、popf 等，这些对计算机的正常运行起着非同小可的影响，操作系统只有亲自执行它们才放心。

另一方面体现在 I/O 读写控制上。IO 读写特权是由标志寄存器 eflags 中的 IOPL 位和 TSS 中的 IO 位图决定的，它们用来指定执行 IO 操作的最小特权级。IO 相关的指令只有在当前特权级大于等于 IOPL 时才能执行，所以它们称为 IO 敏感指令（I/O Sensitive Instruction），如果当前特权级小于 IOPL 时执行这些指令会引发处理器异常。这类指令有 in、out、cli、sti。所以你懂的，不只是操作系统可以进行 IO 端口访问，用户进程也是可以的，只是操作系统不允许用户进程这么做。

平时我们被灌输的思想是用户进程无法直接访问硬件，必须要向操作系统求助，只有高高在上的操作系统才有能力访问外设。操作系统的职责就是管理计算机中的资源，资源包括软件和硬件，不允许用户进

程直接操作外设，这只是操作系统的一种管理策略，因为这是出于对计算机的保护，谁能保证用户程序个个都那么善良可靠呢？万一用户程序非法使用硬件，这种破坏可是难以估量呢，保护计算机安全是操作系统的责任，不应该让不受信任的程序有破坏计算机的可能。

我们在很久以前就介绍过 eflags 寄存器啦，现在来查看下 eflags 寄存器的 IOPL 位，如图 5-60 所示。

31...	21	20	19	18	17	16	15	14	13-12	11	10	9	8	7	6	5	4	3	2	1	0
保留	ID	VIP	VIF	AC	VM	RF		NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF

▲图 5-60　寄存器的 IOPL 位

在 eflags 寄存器中第 12～13 位便是 IOPL（I/O Privilege Level），即 IO 特权级，它除了限制当前任务进行 IO 敏感指令的最低特权级外，还用来决定任务是否允许操作所有的 IO 端口，对，没错，是全部 IO 端口，IOPL 位是打开所有 IO 端口的开关（用来单独设置端口访问的方式是 IO 位图，一会儿介绍）。每个任务（内核进程或用户进程）都有自己的 eflags 寄存器，所以每个任务都有自己的 IOPL，它表示当前任务要想执行全部 IO 指令的最低特权级，也就是处理器最低的 CPL，只有任务的当前特权级大于等于 IOPL 才允许执行全部 IO 指令，即数值上 CPL≤IOPL。

CPL 为 0 时处理器是法力无边的，所以 0 特权级下处理器是不受 IO 限制的。

IOPL 如何设置呢？

用户程序可以在由操作系统加载时通过指定整个 eflags 设置，操作系统如何设置自己的 IOPL 呢？即使内核 IOPL 为 0 也得写进去 eflags 寄存器中才生效。可惜的是没有直接读写 eflags 寄存器的指令，不过可以通过将栈中数据弹出到 eflags 寄存器中来实现修改。可以先用 pushf 指令将 eflags 整体压入栈，然后在栈中修改相应位，再用 popf 指令弹出到 eflags 寄存器中。另外一个可利用栈的指令是 iretd，用 iretd 指令从中断返回时，会将栈中相应位置的数据当成 eflags 的内容弹出到 eflags 寄存器中。所以可以改变 IOPL 的指令只有 popf 和 iretd 指令，依然是只有在 0 特权下才能执行。如果在其他特权级下执行此指令，处理器也不会引发异常，只是没任何反应。

接下来看看 IO 位图是怎么回事。

假如，数值上CPL≤IOPL，程序既可以执行IO特权指令，又可以操作所有的IO端口。倘若数值上CPL＞IOPL，程序也不是完全无法进行任何 IO 操作，有点奇怪是不，似乎和咱们的逻辑不符，其实这样是有道理的。

之前说过，IOPL 是所有 IO 端口的开关，不过，这个开关还留有余地，如果将开关打开，便可以访问全部 65536 个端口，如果开关被关上，即数值上 CPL＞IOPL，则可以通过 IO 位图来设置部分端口的访问权限。也就是说，先在整体上关闭，再从局部上打开。这有点像设置防火墙的规则，先默认为全部禁止访问，想放行哪些端口再单独打开。

处理器为什么允许这么做呢？原因是为了提速。

如果所有 IO 端口访问都要经过内核的话，由低特权级转向高特权级时是需要保存任务上下文环境的，这个过程也是要消耗处理器时间的，随着端口访问多起来，时间成本还是很可观的。这一典型的应用就是硬件的驱动程序，它位于特权级 1。

什么是驱动程序？

驱动程序就是通过 in、out 等 IO 指令直接访问硬件的程序，它为上层程序提供对硬件的控制访问，相当于硬件的代理，程序员通过它就免去了学习硬件控制的相关知识，简化了程序设计。

所以说，驱动程序肯定是要直接控制 IO 端口的，尽管它可以像 Linux 那样位于 0 特权级，但它位于 1 特权级时，依然可以直接操作硬件端口。

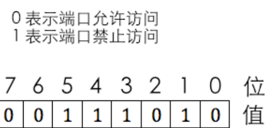
即使是在 3 特权级下，也要考虑某些需要快速反应的场合，比如某个应用程序需要快速的以硬件交互，所以处理器允许通过 I/O 位图来为 3 特权级程序打开某些端口的控制。这些规则同样适用于 2 特权级，也就是说在任意特权级下，处理器都可以通过 I/O 位图为相应特权级的程序开启特定的端口。

欲知 I/O 位图是怎么回事，咱们先把位图的概念明确。

位图就是 bit map，map 就是映射，建立的是某种对应关系，像地图那样，图上某个区域代表实际地

理范围，bit 就是位，bit map 就是用一个 bit 映射到某个实际的对象。位图这种结构的操作单位就是 bit，所以位图其实就是一串二进制 01 数字，对位图的操作也就是读写相应的位，处理器中对内存的访问是以字节为单位的，不能直接操作位，所以对于操作位图，简单说来就是先将该位所在的字节读到内存，若是想将该位置为 1，可以用 1 对该位进行或运算，若想将该位清 0，可以用 0 对该位进行与运算，以后咱们少不了操作位图，到时候再实践。

Intel 处理器最大支持 65536 个端口，它允许任务通过 I/O 位图来开启特定端口，位图中的每一 bit 代表相应的端口，比如第 0 个 bit 表示第 0 个端口，第 65535 个 bit 表示第 65535 个端口，65536 个端口号占用的位图大小是 63356/8=8192 字节，即 8KB。I/O 位图中如果相应 bit 被置为 0，表示相应端口可以访问，否则为 1 的话，表示该端口禁止访问，如图 5-61 所示。



▲图 5-61　位图示意

再次声明，I/O 位图只是在数值上 CPL > IOPL，即当前特权级比 IOPL 低时才有效，若当前特权级大于等于 IOPL，任何端口都可直接访问不受限制。

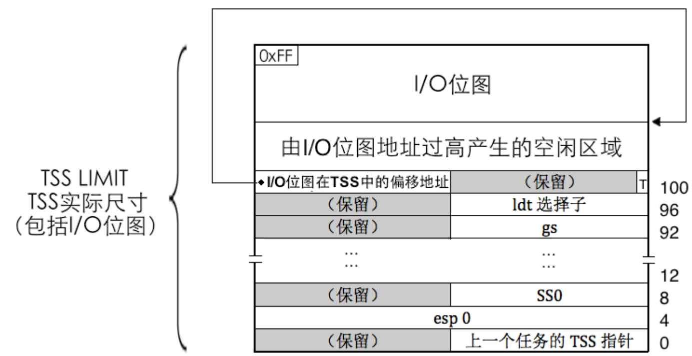
I/O 位图是位于 TSS 中的，它可以存在，也可以不存在，它只是用来设置对某些特定端口的访问，没有它的话便默认为禁止访问所有端口。正是由于它可有可无，所以 TSS 的段界限 TSS limit，即实际大小-1，并不固定。当 TSS 中不包括 I/O 位图时，TSS 只有 104 字节大小。话说回来了，当处理器执行某些 IO 指令时，若当前特权级比 IOPL 低，处理器就会认为也许只是给当前任务单独放行了某些端口，于是它就到 TSS 中找 I/O 位图，如果 I/O 位图不存在，即所有端口都禁止访问，于是处理器就会抛异常。

读到这里，有两个问题要解决。

（1）处理器到 TSS 中哪里去找 I/O 位图呢？

（2）怎样证明 I/O 位图不存在？

在图 5-47 所示 TSS 结构中，有一项是“I/O 位图在 TSS 中的偏移地址”，它在 TSS 中偏移 102 字节的地方，占 2 个字节空间，就是图 5-47 的左上角，此处用来存储 I/O 位图的偏移地址，即此地址是 I/O 位图在 TSS 中以 0 为起始的偏移量。如果某个 TSS 存在 I/O 位图的话，此处用来保存它的偏移地址，示意如图 5-62 所示。



▲图 5-62　TSS 中的 I/O 位图

如图 5-62 所示，TSS 中如果有 I/O 位图的话，它将位于 TSS 的顶端，这就是 TSS 的实际尺寸并不固定的原因，当包括 I/O 位图时，其大小是“I/O 位图偏移地址”+8192+1 字节，结尾这个 1 字节是 I/O 位图中最后的 0xff，说来话长，一会再解释。若不包括 I/O 位图，其大小则为最小尺寸 104 字节。由于 I/O 位图偏移地址并不固定，可以大于等于 104，所以在 TSS 中偏移 102 字节和 I/O 位图之间可能会有空闲区域。

您看，既然 I/O 位图位于 TSS 内，那它的地址必须是在 TSS 的尺寸范围之内，即地址的范围是在 TSS

偏移（104 ～TSS 段界限 limit）之间，如果偏移地址不在此范围，即大于等于 TSS 段界限 limit（TSS 尺寸大小-1），则表明没有 I/O 位图。

现在来说下为什么在 IO 位图的结尾有个 0xff。

在计算机系统硬件中，IO 端口是按字节来编址的，意思是说一个端口只能读写 1 个字节的数据。如果对一个端口连续读写多个字节，实际上是从以该端口号为起始的多个端口一并读进来的。举个例子，比如 in 指令可以读取 16 位端口数据，即一次读取 2 字节，假设端口 0x234 是 16 位端口：

```
in ax,0x234
```

这相当于

```
in al,0x234
in ah,0x235
```

处理器在进行端口读写时，若当前特权级 CPL 低于 IO 特权级 IOPL 时，如果有 I/O 位图的话，处理器会在 I/O 位图中检查端口相应的 bit 是否为 0。若在某个端口中读取多个字节，处理器必然会检查连续的多个端口在 I/O 位图中对应的多个 bit，这些 bit 必须都得为 0 才允许访问它们。

连续的多个 bit 也许会跨字节，比如端口 0x234 对应的 bit 在前一个字节的最后一位，0x235 对应的 bit 在后一个字节的第 0 位，这样处理器必须将这两个字节都读进来处理。

大多数情况下跨字节都没问题，但当第 1 个 bit 在位图的最后一个字节时就会出问题，处理器要读进多个字节，所以，第 2 个 bit 所在的字节就越界了，该字节已经不属于位图范围。

为解决这个问题，处理器要求位图的最后一字节必须是 0xFF，此字节有两个作用。

第一，处理器允许 I/O 位图中不映射所有的端口，即 I/O 位图长度可以不足 8KB，但位图的最后一字节必须为 0xFF。如果在位图范围外的端口，处理器一律默认禁止访问。这样一来，如果位图最后一字节的 0xFF 属于全部 65536 个端口范围之内，字节各位全为 1 表示禁止访问此字节代表的全部端口，这并没什么过错。

第二，如果该字节已经超过了全部端口的范围，它并不用来映射端口，只是用来作为位图的边界标记，用于跨位图最后一个字节时的“余量字节”。避免越界访问 TSS 外的内存。

这就是位图中最后一字节必须为 0xFF 的原因。

您看 I/O 位图我说得这么热闹，其实咱们不用它，这里的介绍完全是为了让大伙了解 IO 访问的工作原理，只会有益无害。

到了这里，特权级就讲完了，本章也结束了，友情提示，我知道本章内容有点多，您已经很累了，现在所说的是结束语，所以可以略过这里不看啦。虽然我们已经在内核中，但这种看不到的成果似乎没有说服力，咱们在下一章中让大家看到工作中的内核。这不仅涉及一些学过的知识，而且还有新的东西要带给大家，大家要养足精神才好。

第 6 章　完善内核

上一章中我们终于完成了内核，但该内核确实没法再简陋了，它啥都没做，所以，从本章开始我们要对它悉心养育，逐渐丰富它的功能。

6.1 函数调用约定简介

由于我们要将 C 语言和汇编语言结合编程，所以一定会存在汇编代码和 C 代码相互调用的问题，有些事情还是要提前交待给大家的，本节就是要给大家说下函数调用规约中的那些事儿。

函数调用约定是什么？

调用约定，calling conventions，从字面上理解，它是调用函数时的一套约定，是被调用代码的接口，它体现在：

- 参数的传递方式，是放在寄存器中？栈中？还是两者混合？
- 参数的传递顺序，是从左到右传递？还是从右到左？
- 是调用者保存寄存器环境，还是被调用者保存？保存哪些寄存器呢？

我估计，我这么解释调用约定的话，之前对此不懂的同学还是不懂，所以咱们得从头说起啦。没例子还真说不清楚，咱们还是拿例子来说事吧。

比如在 C 语言中我们有这样的代码：

```
int subtract(int a, int b) {
return a-b;
}
```

我们可以用这样的形式调用它：

```
int sub = subtract(3,2);
```

这样 sub 的值就变成了 1。这是我们司空见惯的用法，但大家有没有想过，计算机是如何确定参数 3 和 2 在哪里的呢？这是有关参数存储的问题。

计算机中可没有专门存储参数的硬件，即使有的话，我想也不太容易确定该硬件的容量，毕竟参数的个数是不定的。而且还有个致命的问题，若在刚刚传入参数之后，函数执行之前被换下了 CPU，新的进程上 CPU 后，也要调用函数，也要传递参数呢，还是会引出参数覆盖的问题。不过咱们之前说过，参数可以放在寄存器中，也可以放在内存中。

寄存器数量是有限的，假设将参数放在寄存器中传递的话，主调函数必然要考虑保存寄存器现场的问题，一是用哪些寄存器传参数，二是用于传递参数的寄存器，其原来的值如果要保留的话，往哪里保存呢？估计大家也是这么想的，内存足够大，肯定是往内存中转存啦，那既然是还要在内存中折腾，不如直接把参数放在内存中更直接省事。

说到用内存来传递参数，还要考虑内存地址，用哪块内存来存储参数呢？为了避免多进程的参数覆盖问题，每个进程的参数得单独存储在不同地址，得在内存中再为每个进程规划出一块存储参数的内存区域，想想就很麻烦。或许您早已经迫不及待想说出答案啦：栈也是位于内存中的啊，最好的方式就是在栈中来保存。这有两个好处。

（1）首先，每个进程都有自己的栈，这就是每个内存自己的专用内存空间。

（2）其次，保存参数的内存地址不用再花精力维护，已经有栈机制来维护地址变化了，参数在栈中的

位置可以通过栈顶的偏移量来得到。

好啦，参数存储的问题解决了，我们决定在进程自己的栈空间中保存参数，一种可行的方案是调用者在调用函数时，先把所有参数压栈，然后再调用函数。被调用函数在栈中获取到参数后进行处理。

以上方案如果不细想的话似乎还挺好，其实解决了一个问题后，又引入了两个新的问题。

（1）参数若在栈中保存，由谁来负责回收参数所占的栈空间？

（2）当参数很多的情况下，主调函数将参数以什么样的顺序传递呢？因为这决定被调用函数获取参数的准确性。

上面提到的回收栈空间或者清理栈空间，并不是把参数在栈中所占据的内存清 0，而是回收参数所在的内存空间，也就是让栈顶恢复到栈中参数所在的位置之前，即让栈指针往高地址处回退。这样一来，参数原本占用的空间又变得可用了，下次再有入栈操作时，push 指令可以直接将其覆盖。

也许有部分同学并未意识到这两个问题，心想，我自己写的函数，我自己调用，难道我自己还不知道怎么处理吗？您看，这里用了三个“我自己”来强调问题的关键所在，自己调用自己的代码确实可以避免以上两个问题，只要自己协调好了就一切 ok，可保不准您会调用其他同事写的函数。

调用约定是为解决汇编语言的问题才提出的，不像咱们平时所用的高级语言，直接用实参往函数中一代入就算调用完成了，高级语言中本身不存在这两个问题，高级语言编译器为了方便程序员，默默承担了这些，这两个问题是高级语言在被编译为底层汇编语言时才有的，所以高级语言中不涉及调用约定。

在 C 语言中，咱们不用考虑这些问题，还是拿前面说过的减法函数举例。

```
subtract(int a, int b) {        //被调用者
    return a-b;
}
int sub = subtract(3,2);        //主调用者
```

函数 subtract 返回 a 减 b 的差，这里只要代入实参 3 和 2 即可完成调用。可是，在其被编译为汇编语言时，参数是要压入栈中的，现在问题来了。我们模拟一下这种情况，以上 c 代码中的调用方和被调用方对应的汇编代码如下：

主调用者：

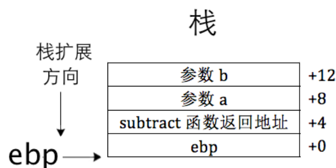
```
1 push 2                   ;压入参数 b
2 push 3                   ;压入参数 a
3 call subtract            ;调用函数 subtract
```

被调用者：

```
1 push ebp                 ;备份 ebp，为以后用 ebp 作为基址来寻址参数
2 mov ebp, esp             ;将当前栈顶赋值给 ebp
3 mov eax, [ebp+8]         ;得到被减数，参数 a
4 sub eax,[ebp+12]         ;得到减数，参数 b
5 pop ebp                  ;恢复 ebp 的值
…
```

目前栈中的情况如图 6-1 所示。

如果调用者和被调用者（subtract 函数）都是同一个程序员写的，他很清楚自己压入栈中参数的顺序，所以他在 subtract 函数中，明确地知道栈中[ebp+8]处的内容是被减数 a，[ebp+12]处的是减数 b。其实，这个程序员在潜意识中自己跟自己建立了个约定，先被压入栈的是减数 b，后被压入的是被减数 a，这样他才能确信从容地在 subtract 函数体中获取到正确的参数。其实他也可以反着来，先把被减数 a 压入栈，再把减数 b 压入栈，这样在 subtract 函数中通过[ebp+8]得到的是参数 b（减数），[ebp+12]得到的是参数 a（被减数）。总之参数很多的情况下就会涉及到参数传递的顺序问题，即使是自己负责传递参数的话，也很少有人会今天一个“这样”的顺序传递参数，明天一个“那样”的顺序传递参数。因此参数传递的顺序应该是始终如一的，要么从左到右，要么从右到左，只能选择一种。



▲图 6-1　栈中布局

以上是自己调用自己代码的情况，怎么说都比较方便。可万一，被调用函数 subtract 不是自己写的，咱们不知道在 subtract 把[ebp+8]当作被减数 a，还是减数 b，咱们该以怎样的顺序将参数压入栈中呢？这得跟人家商量了，双方得协调个大家认同的参数入栈顺序，这就是最初调用约定的由来。

我们要解决的不只是参数压栈顺序问题，还有栈空间的清理工作呢。其实问题倒也不难解决，这都是属于调用方和被调用方之间协调的问题，只要双方提前商量好传入参数的顺序和由谁来负责清理栈空间就行。

在高级语言中，这两个问题是通过调用约定来解决的，调用约定就是调用方和被调用方就以上问题达成一致解决方案的约定，双方按照这种约定合作就不会发生问题。我们按照由谁来清理栈空间分类，目前的调用约定见表 6-1。

表 6-1　调用约定简述

栈清理负责人	类　别	描　述
调用者清理	cdecl	cdecl（C declaration，即 C 声明），起源于 C 语言的一种调用约定，在 C 语言中，函数参数是从右到左的顺序入栈的。GNU/Linux GCC，把这一约定作为事实上的标准，x86 架构上的许多 C 编译器也都使用这个约定。在 cdecl 中，参数是在栈中传递的。EAX、ECX 和 EDX 寄存器是由调用者保存的，其余的寄存器由被调用者保存。函数的返回值存储在 EAX 寄存器。由调用者清理栈空间
	syscall	与 cdecl 类似，参数从右到左入栈。参数列表的大小被放置在 AL 寄存器中。syscall 是 32 位 OS/2 API 的标准
	optlink	参数也是从右到左压栈。从最左边开始的三个参数会被放置在寄存器 EAX, EDX 和 ECX 中，最多四个浮点参数会被传入 ST（0）到 ST（3）中，虽然这四个参数的空间也会在参数列表的栈上保留。函数的返回值在 EAX 或 ST（0）中。保留的寄存器有 EBP, EBX, ESI 和 EDI。optlink 在 IBM VisualAge 编译器中被使用
被调用者清理	pascal	基于 Pascal 语言的调用约定，参数从左至右入栈（与 cdecl 相反）。被调用者负责在返回前清理堆栈。此调用约定常见在如下 16-bit API 中：OS/2 1.x，微软 Windows 3.x，以及 Borland Delphi 版本 1.x
	register	Borland fastcall 的别名
	stdcall	这是一个 Pascal 调用约定的变体，被调用者依旧负责清理堆栈，但是参数从右往左入栈——与 cdecl 一致。寄存器 EAX, ECX 和 EDX 被指定在函数中使用，返回值放置在 EAX 中。stdcall 对于微软 Win32 API 和 Open Watcom C++是标准
	fastcall	此约定还未被标准化，不同编译器的实现也不一致。典型的 fastcall 约定会传递一个或多个参数到寄存器上，以减少对内存的访问
	Microsoft fastcall	Microsoft 或 GCC 的__fastcall 约定，也即__msfastcall，传入头两个参数（从左至右）到 ECX 和 EDX 中，剩下的参数从右至左压栈
被调用者清理	Borland fastcall	从左至右，传入三个参数至 EAX，EDX 和 ECX 中。剩下的参数入栈，也是从左至右。在 32 位编译器 Embarcadero Delphi 中，这是默认的调用约定，在编译器中以 register 形式为人知。在 i386 上的某些版本 Linux 也使用了此约定
调用者或被调用者清理	thiscall	在调用 C++非静态成员函数时使用此约定。基于所使用的编译器和函数是否使用可变参数，有两个主流版本的 thiscall。对于 GCC 编译器，thiscall 几乎与 cdecl 等同：调用者清理堆栈，参数从右到左传递。差别在于 this 指针，thiscall 会在最后把指针推入栈中，虽然在函数原型中它是隐式的第一个参数。在微软 Visual C++编译器中，this 指针被传到 ECX 寄存器上，被调用者负责清理堆栈，其余同此编译器的 C 版本和 Windows API 函数使用的 stdcall 约定。当函数使用可变参数，此时调用者负责清理堆栈（参考 cdecl）。 thiscall 约定只在微软 Visual C++ 2005 及其之后的版本被显式指定。其他编译器中，thiscall 并不是一个关键字（反汇编器，如 IDA 使用__thiscall）

以上信息是我在 wiki 中摘录汇总的，并非原创，所以我也没有能力将每种调用约定都给大家说明白。这里还是本着“够用就行”的原则，咱们用了哪种就介绍哪种，因为 C 语言遵循的调用约定是 cdecl，咱们也自然要遵守 cdecl 约定了。不过为了起到对比的作用，除了介绍 cdecl 外，也会介绍下 stdcall。

既然咱们用的调用约定是 cdecl，那对它的介绍最好让它离下一节的内容近一些，所以先说一下咱们不用的 stdcall，其实这两个差别就在于由谁来回收栈空间。

stdcall 的调用约定意味着。

（1）调用者将所有参数从右向左入栈。

（2）被调用者清理参数所占的栈空间。

这两点在表 6-1 的介绍中大家已有所了解，下面咱们将理论实践一下，还是拿上面说过的函数举例。

```
1 int subtract(int a, int b);   //被调用者
2 int sub = subtract (3,2);     //主调用者
```

第 1 行是个函数声明，其实现已经在前面看到了，就是“return a-b”。

第 2 行进行函数调用，实参分别是 3 和 2。在实际调用中，参数按照从右向左的顺序，参数 b 会先被压入栈，然后是参数 a 压入栈。在 stdcall 调用约定下，这个 c 代码被编译后的汇编语句是：

主调用者：

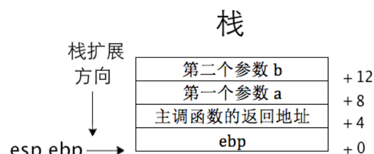
```
; 从右到左将参数入栈
1 push 2                        ;压入参数 b
2 push 3                        ;压入参数 a
3 call subtract                 ;调用函数 subtract
```

以上是主调函数，现在看下被调函数 subtract 中做了什么。

被调用者：

```
1 push ebp                      ;压入 ebp 备份
2 mov ebp,esp                   ;将 esp 赋值给 ebp
                                ;用 ebp 作为基址来访问栈中参数
3 mov eax,[ebp+0x8]             ;偏移 8 字节处为第 1 个参数 a
4 add eax,[ebp+0xc]             ;偏移 0xc 字节处是第 2 个参数 b
                                ;参数 a 和 b 相加后存入 eax
5 mov esp,ebp                   ;为防止中间有入栈操作,用 ebp 恢复 esp
                                ;本句在此例子中可有可无,属于通用代码
6 pop ebp                       ;将 ebp 恢复
7 ret 8                         ;数字 8 表示返回后使 esp+8
                                ;函数返回时由被调函数清理了栈中参数
```

当执行流进入到 subtract 后，在它的内部为了用 ebp 作为基址引用栈中参数，先执行了 push ebp 来备份 ebp，再将栈指针赋给了 ebp。目前栈中布局如图 6-2 所示。



▲图 6-2 进入 subtract 函数后栈中布局

大家根据图 6-2 很容易地看出 ebp 偏移 8 字节是参数 a，偏移 12 字节是参数 b。以上代码值得说一下的是 ret 8 这句。stdcall 是被调用者负责清理栈空间，这里的被调用者是函数 subtract。也就是说，subtract 需要在返回前或返回时完成。在返回前清理栈相对困难一些，清理栈是指将栈顶回退到参数之前。因为返回地址在参数之下，ret 指令执行时必须保证当前栈顶是返回地址。所以通常在返回时“顺便”完成。于是 ret 指令便有了这样的变体，其格式为：

ret 16 位立即数

这是允许在返回时顺便再将栈指针 esp 修改的指令。顺便说一句，由于 32 位下 push 指令不是压入字，就是压入双字，所以 ret 的参数必须是偶数。在 ret 8 执行之前，当前栈顶必须是返回地址，即使没有第 5 行的代码，当前 esp 也等同于 ebp，因为之前没有任何 push 压栈操作，这是编译器为了通用性而加进去的，所以我们在注释中写到，此句可有可无。在经过第 6 行将栈顶（当前 esp 指向的内存）弹出到 ebp 之后，ebp 被恢复，此时 esp 指向了+4 字节的位置，即当前栈顶为主调函数的返回地址。结合图 6-2，ret 指令将栈顶的数据弹出到寄存器 eip 后，栈指针 esp 自加 4，由于还有个参数 8，所以 esp 又被加了 8，从而跳过了参数 a 和 b，顺利地完成了被调用者清理栈的任务。

stdcall 是调用者在栈中压入参数，由被调用者回收栈空间。貌似分工很明确，配合很默契。因为被调用者知道自己需要几个参数，所以知道要回收多少栈空间。但转念一想，凡事都要自己亲力亲为才放心，调用者压入的参数，万一被调用者忘记回收栈空间该怎么办（这一点由高级语言编译器保证，一般不会，大伙儿放心，本段这么写是为了表述下一种调用约定方式的特点），参数多了栈会溢出的。下面咱们就要介绍这种“亲力亲为”的调用约定，即调用者自己向栈中压入参数，还是由调用者自己回收栈空间。

好啦，stdcall 调用约定就到此为止，下面咱们聊聊 cdecl 调用约定，这才是咱们所使用的调用规范。

cdecl 调用约定由于起源于 C 语言，所以又称为 C 调用约定，是 C 语言默认的调用约定。cdecl 的调用约定意味着。

（1）调用者将所有参数从右向左入栈。

（2）调用者清理参数所占的栈空间。

您也看到了，它和 stdcall 一样都是从右向左将参数入栈的，区别就是 cdecl 由调用者清理栈空间。这就是刚才所说的“亲力亲为”的调用约定。也许您会说，难道调用者就不会忘记清理栈空间吗（编译器一般不会忘）？哈哈，也许也会吧，不过既然调用者和被调用者都有可能忘记清理栈，cdecl 至少把控了栈的控制权，这不是也很好吗。

其实，cdecl 调用约定最大的亮点是它允许函数中参数的数量不固定，我们熟识的 printf 函数，它能够支持变长参数，就是利用此 cdecl 调用约定的性质设计出来的，它的原理是利用字符串参数 format 中的'%'来匹配栈中的参数，以后咱们在动手实现 printf 函数时会体验到这一优势。

好啦，上菜啦，咱们看看在 cdecl 调用约定下产生的汇编代码，这里还是拿之前的 subtract 函数举例。

```
1 int subtract(int a, int b);   //被调用者
2 int sub = subtract (3,2);     // 主调用者
```

主调用者：

```
; 从右到左将参数入栈
1 push 2                        ;压入参数 b
2 push 3                        ;压入参数 a
3 call subtract                 ;调用函数 subtract
4 add esp, 8                    ;回收（清理）栈空间
```

被调用者：

```
1 push ebp                      ;压入 ebp 备份
2 mov ebp,esp                   ;将 esp 赋值给 ebp
                                ;用 ebp 作为基址来访问栈中参数
3 mov eax,[ebp+0x8]             ;偏移 8 字节处为第 1 个参数 a
4 add eax,[ebp+0xc]             ;偏移 0xc 字节处是第 2 个参数 b
                                ;参数 a 和 b 相加后存入 eax
5 mov esp,ebp                   ;为防止中间有入栈操作，用 ebp 恢复 esp
                                ;本句在此例子中可有可无,属于通用代码
6 pop ebp                       ;将 ebp 恢复
7 ret
```

和 stdcall 相比，在 cdecl 调用约定下生成的汇编代码，就是在被调用者中的回收栈空间操作挪到了主调用者中，在主调用者代码中的第 4 行，通过将 esp 加上 8 字节的方式回收了参数 a 和参数 b，本例中的其他代码都和 stdcall 一样。

好啦，说完 cdecl 后，有关调用约定的内容咱们也介绍完了，现在咱们该去写内核啦。

6.2 汇编语言和 C 语言混合编程

本来说好的接下来的工作是要去“丰满”我们的内核，可咱们这种一步一回头的学习方式还得继续啊。其实我了解大家急切写内核的心情，但本书的目的不是写一个操作系统就完事了，而是让大家明白一个至少能运行的操作系统为什么要这样写，所以咱们的学习方式必然是边学习理论知识边实践。如果不给大家交待清楚必要的理论知识，我也对不起自己的良心，我不能为了自己的懒惰而假装大家都明白了。另外，既然咱们都渴望学习，能了解到更多的混合编程方式不是更好吗？

6.2.1　浅析 C 库函数与系统调用

开门见山，汇编语言和 C 语言混合编程可分为两大类。

（1）单独的汇编代码文件与单独的 C 语言文件分别编译成目标文件后，一起链接成可执行程序。

（2）在 C 语言中嵌入汇编代码，直接编译生成可执行程序。

本节所说的“汇编语言和 C 语言混合编程”属于第 1 种，第 2 种的内嵌汇编又称为内联汇编，以后咱们会有专门的章节来说的。在内核文件中，有些比较长的汇编代码真不适合用内联汇编完成，还是需要

专门写个汇编代码文件专项专用。

简单起见，咱们先学习下 Linux 系统调用，利用系统调用能够帮助简化演示的模型。

系统调用是 Linux 内核提供的一套子程序，它和 Windows 的动态链接库 dll 文件的功能一样，用来实现一系列在用户态不能或不易实现的功能，比如最常见的读写硬盘文件，只有操作系统有权限去访问硬件，用户程序是没有权限的，用户程序只能向操作系统寻求帮助，故系统调用是供用户程序来使用的，操作系统权利至高无上，不需要使用自己对外发布的功能接口，即系统调用。

由于是用户程序想使用操作系统提供的功能，所以系统调用又称为操作系统功能调用。

系统调用很像 BIOS 中断调用（在很久很久以前咱们有说过 BIOS 中断、DOS 中断等内容），只不过系统调用的入口只有一个，即第 0x80 号中断，它不像 BIOS 中断那样，几乎一个功能就有一个入口，所以您在 BIOS 中断手册中会见到那么多的中断调用啦，比如中断号 0～0x20 都是 BIOS 的中断调用。

为什么系统调用只有一个入口呢？以后咱们学习中断机制的时候就会明白，中断的实现是要用到中断描述符表的，表中很多中断项（号）是被预留的，不能强占，所以 Linux 就选了一个可用的中断号作为所有系统调用的统一入口，具体的子功能在寄存器 eax 中单独指定。

总之，BIOS 中断走的是中断向量表，所以有很多中断号给它用，而系统调用走的是中断描述符表中的一项而已，所以只用了第 0x80 项中断。

系统调用的子功能要用 eax 寄存器来指定，所以咱们要看看有哪些系统调用啦，在 Linux 系统中，系统调用是定义在/usr/include/asm/unistd.h 文件中，该文件只是个统一的入口，指向了 32 位和 64 位两种版本。在 asm 目录下提供了这两个版本，文件名分别是 unistd_32.h 和 unistd_64.h，这里给大家摘录了部分 32 位 x86 平台下的 unistd_32.h 文件，如图 6-3 所示。

在/usr/include/asm/unistd_32.h 文件中共定义了 348 个系统调用，哦，给大家说一下，我用的 Linux 版本是 CentOS release 6.3 （Final），不知道新版本内核中是否增加了新的系统调用功能。

我们要用的系统调用是第 4 号调用，即__NR_write。不要被它前面的两个下画线吓到，就是个命名而已，它代表我们所说的 write 系统调用。

如果不知道某个系统调用的用法，可以用 man 命令来查看，方法是 man 2 系统调用名。咱们执行 man 2 write 看看，如图 6-4 所示。

```
 1 #ifndef _ASM_X86_UNISTD_32_H
 2 #define _ASM_X86_UNISTD_32_H
 3
 4 /*
 5  * This file contains the system call numbers.
 6  */
 7
 8 #define __NR_restart_syscall      0
 9 #define __NR_exit                 1
10 #define __NR_fork                 2
11 #define __NR_read                 3
12 #define __NR_write                4
13 #define __NR_open                 5
14 #define __NR_close                6
15 #define __NR_waitpid              7
16 #define __NR_creat                8
17 #define __NR_link                 9
18 #define __NR_unlink              10
19 #define __NR_execve              11
20 #define __NR_chdir               12
21 #define __NR_time                13
22 #define __NR_mknod               14
23 #define __NR_chmod               15
```

▲图 6-3　部分 Linux 系统调用

```
NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write() writes up to count bytes from the buffer pointed
buf to the file referred to by the file descriptor fd.
```

▲图 6-4　write 系统调用说明

man 后面的数字 2 表示查看 System Calls 方面的帮助，对于 man 自己的帮助信息，man 命令也可以 man 自己，可以用 man man 来查看。

图 6-4 所示只是部分帮助信息，咱们了解这些就够用了。write 的功能是把 buf 指向的缓冲区中的 count 个字节写入 fd 指向的文件描述符，执行成功后返回写入的字节数，失败则返回-1。

如果在 C 语言中调用 write 的话，直接代入实参就行了，这是最简单的方式，如代码 c_syscall.c：

```
#include <unistd.h>
int main(){
write(1,"hello,world\n",4);
return 0;
}
```

为了使用 c 标准库中的 write 函数，文件开头包含了标准头文件 unistd.h，通过该函数可以使用系统的 write 系统调用，该文件在磁盘上的路径是/usr/include/unistd.h。不过在本机上测试发现不包含 unistd.h，其编译、运行都没问题，也许这和隐式声明有关，这里不再深究。

调用“系统调用”有两种方式。

（1）将系统调用指令封装为 c 库函数，通过库函数进行系统调用，操作简单。

（2）不依赖任何库函数，直接通过汇编指令 int 与操作系统通信。

以上的 c 代码就是用的第一种方式，不知道您是否对 write 函数的内部实现感兴趣，其实我也没研究过，不过万变不离其宗，核心思想是必须进行内核沟通才能获得内核提供的功能。所以，write 内部封装的一定是系统调用指令，按照这种设想，一会咱们会模拟一下它的实现。

我们这里要介绍下第二种：跨过库函数直接与系统内核通信，这样最终的程序不需要与任何库文件链接，这是获得系统功能效率最高的方式。

我相信，如果曾经学过汇编语言，老师都给咱们演示过第二种方式，但大多数同学还是觉得云里雾里，即使照葫芦画瓢完成了打印字符串的工作，也有部分同学不清楚自己在做什么，所以我在这里尽量多说一点。

前面我们已经知道了 write 系统调用函数的 C 语言使用方式，我们要用汇编代码直接与内核通信该怎么做？我们要看看系统调用输入参数的传递方式。

当输入的参数小于等于 5 个时，Linux 用寄存器传递参数。当参数个数大于 5 个时，把参数按照顺序放入连续的内存区域，并将该区域的首地址放到 ebx 寄存器。这里我们只演示参数小于等于 5 个的情况。

eax 寄存器用来存储子功能号（寄存器 eip、ebp、esp 是不能使用的）。5 个参数存放在以下寄存器中，传送参数的顺序如下。

（1）ebx 存储第 1 个参数。

（2）ecx 存储第 2 个参数。

（3）edx 存储第 3 个参数。

（4）esi 存储第 4 个参数。

（5）edi 存储第 5 个参数。

好啦，理论知识够用啦，现在赶紧实践一把，见以下代码 syscall_write.S

```
 1 section .data
 2 str_c_lib: db "c library says: hello world!", 0xa  ;0xa 为 LF ASCII 码
 3 str_c_lib_len equ $-str_c_lib
 4
 5 str_syscall: db "syscall says: hello world!", 0xa
 6 str_syscall_len equ $-str_syscall
 7
 8 section .text
 9 global _start
10 _start:
11 ;;;;;;;;;;;;;  方式 1: 模拟 C 语言中系统调用库函数 write ;;;;;;;;;;;;;
12    push str_c_lib_len        ;按照 C 调用约定压入参数
13    push str_c_lib
14    push 1
15
16    call simu_write           ;调用下面定义的 simu_write
17    add esp,12                ;回收栈空间
18
19 ;;;;;;;;;;;;;  方式 2: 跨过库函数，直接进行系统调用 ;;;;;;;;;;;;;
20    mov eax,4                 ;第 4 号子功能是 write 系统调用（不是 C 库函数 write）
21    mov ebx, 1
22    mov ecx, str_syscall
23    mov edx, str_syscall_len
24    int 0x80                  ;发起中断，通知 Linux 完成请求的功能
```

```
25
26 ;;;;;;;;;;;;; 退出程序 ;;;;;;;;;;;
27     mov eax,1                ;第 1 号子功能是 exit
28     int 0x80                 ;发起中断，通知 Linux 完成请求的功能
29
30 ;;;;;;;下面自定义的 simu_write 用来模拟 C 库中系统调用函数 write
   ;;;;;;这里模拟它的实现原理
31 simu_write:
32     push ebp                 ;备份 ebp
33     mov ebp,esp
34     mov eax,4  ;第 4 号子功能是 write 系统调用（不是 C 库函数 write）
35     mov ebx, [ebp+8]         ;第 1 个参数
36     mov ecx, [ebp+12]        ;第 2 个参数
37     mov edx, [ebp+16]        ;第 3 个参数
38     int 0x80                 ;发起中断，通知 Linux 完成请求的功能
39     pop ebp                  ;恢复 ebp
40     ret
```

代码 syscall_write.S 中，我们演示了系统调用的两种方式。程序开头定义了两种方式下打印的字符串，其中 0xa 为 LF（LineFeed）ASCII 码，这样就会输出一个换行符。

第 11～17 行是在演示方式 1，模拟调用 C 库函数 write 的方式。因为 write 是 C 库函数，按一般的做法是汇编程序需要与 C 代码生成的目标文件链接才能调用 C 的代码。在这个例子中我们并没有这样做，因为我想让大家了解 write 函数的本质，所以，在这里为大家定义了 simu_write 来代替 C 库函数 write，用它来简单解释 write 的原理，它定义在第 1～40 行。这里是按照 C 调用约定将参数从右到左依次入栈，随后调用 simu_write 实现字符串打印功能。

第 19～24 行是在演示第 2 种系统调用的方式，这是最简单直接可依赖的方式。第 0～24 行是在 eax 中赋予子功能号，参数按照顺序依次写入对应的寄存器。

第 31～40 行是 simu_write 的实现，它内部在本质上和第 2 种方式一样，都是在内部调用 int 指令直接和系统通信实现系统调用。此函数只是为了试图揭开 C 库函数的实现原理，良苦用心您懂的。

好啦，编译链接过程如下：

```
nasm -f elf -o syscall_write.o syscall_write.S
```

其中-f 参数用来指定编译输出的文件格式，这里需要指定为 elf，目的是将来要和 gcc 编译的 elf 格式的目标文件链接，所以格式必须相同。nasm 输出为目标文件，已经用-o 指定文件名为 syscall_write.o。

最后用 ld 程序将 syscall_write.o 链接成 elf 格式的二进制可执行文件。

```
ld -o syscall_write.bin syscall_write.o
```

程序执行后的效果如图 6-5 所示。

```
[work@localhost book]$ nasm -f elf -o syscall_write.o syscall_write.S
[work@localhost book]$ ld -o syscall_write.bin syscall_write.o
[work@localhost book]$ ./syscall_write.bin
c library says: hello world!
syscall says: hello world!
[work@localhost book]$
```

▲图 6-5　模拟 C 库函数 write

顺便说一句，syscall_write.bin 如果因为权限不足而无法执行时，可以用以下指令增加执行权限。

```
chmod u+x syscall_write.bin
```

6.2.2　汇编语言和 C 语言共同协作

由于有了上一节的铺垫，本节的内容相对较少，这里给大家准备了两个小文件来实例演示汇编语言和 C 语言相互调用。

会两种不同语言的人，只是掌握了同一件事物的两种表达方式。人在学习一种新语言时，潜意识里是建立了语言符号与事物形象的映射关系，比如我们在学习 grape 这个单词时，我们之所以认为它就是我们所认知的葡萄，是因为我们知道这两个名词都是在描述同一种圆圆的、黑紫色、甜酸的这一水果的形象，如果脑子中不存在这个形象的话，不光是学不会 grape 这个英文单词，就连中文的葡萄也不知道是何意。总之，对于具体的事物，一定是先有其形象，再有其描述，这样才能理解该事物，了解了事物的本质形象后，无论该事物的名字怎样变化，我们都能将它们相互转换。

也许有同学会问，以上这些所说的目的是什么？

“汇编语言和 C 语言可以互相调用”，这句话并不是如表面陈述的那样，似乎是两种语言能直接交流，其

实并不是这样。C 语言和汇编语言完全是不同的东西，它们怎么能认识对方呢。这就像跟不懂汉语的人说汉语，那人听了肯定会晕头转向的，除非身边有个翻译帮忙转述，这个翻译所做的工作实质上是在脑子中找到这种语言所描述的事物形象，然后给出这种事物形象的另一种语言表达，这个事物形象才是翻译的核心。这有些类似上面提到的葡萄的例子，在同一种指令集上的各种计算机程序语言，最终也要编译为那些相同的机器码，这些机器码便是高级语言的本质形象。对于上面提到的翻译，在计算机世界里，就是编译器，只不过这个翻译有多个，例如本书所说的 C 语言编译器 gcc 和汇编语言编译器 nasm，它们能在一起配合，是因为它们都懂机器语言。举个例子，就像小明只会汉语和英语，小红只会汉语和法语，若他们之间在交流时，小明说英语，小红说法语，他俩相互都听不懂，所以，当说英文的小明想跟说法语的小红借作业时，他必须用汉语告诉小红。

编译器知道高级语言所描述的事物形象是机器码，所以各种编译在高级语言方面的交流，本质上是将它们都变成统一的机器码后实现的。

不知道我表达清楚了没有，这里给读者准备了两个小文件：C_with_S_c.c 和 C_with_S_S.S。大家快速浏览一下即可，在代码后面我会讲解。

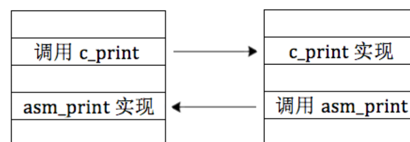
代码 C_with_S_c.c

```
1 extern void asm_print(char*,int);
2 void c_print(char* str) {
3    int len=0;
4    while(str[len++]);
5    asm_print(str, len);
6 }
```

代码 C_with_S_S.S

```
 1 section .data
 2 str: db "asm_print says hello world!", 0xa, 0
 3 ;0xa 是换行符,0 是手工加上的字符串结束符\0 的 ASCII 码
 4 str_len equ $-str
 5
 6 section .text
 7 extern c_print
 8 global _start
 9 _start:
10 ;;;;;;;;;;;; 调用 c 代码中的函数 c_print ;;;;;;;;;;;
11    push str             ;传入参数
12    call c_print         ;调用 c 函数
13    add esp,4            ;回收栈空间
14
15 ;;;;;;;;;;;;;;;;;;; 退出程序 ;;;;;;;;;;;;;;;;;;;
16    mov eax,1           ;第 1 号子功能是 exit 系统调用
17    int 0x80            ;发起中断，通知 Linux 完成请求的功能
18
19 global asm_print       ;相当于 asm_print（str,size）
20 asm_print:
21    push ebp            ;备份 ebp
22    mov ebp,esp
23    mov eax,4           ;第 4 号子功能是 write 系统调用
24    mov ebx, 1          ;此项固定为文件描述符 1，标准输出（stdout）指向屏幕
25    mov ecx, [ebp+8]    ;第 1 个参数
26    mov edx, [ebp+12]   ;第 2 个参数
27    int 0x80            ;发起中断，通知 Linux 完成请求的功能
28    pop ebp             ;恢复 ebp
29    ret
```

代码 C_with_S_c.c 中的函数 c_print 是被汇编代码 C_with_S_S.S 调用的，在 c_print 的实现中，它又调用汇编代码中的 asm_print。它们的关系如图 6-6 所示。



▲图 6-6　汇编代码与 C 代码相互调用

有了这样的全局印象后，我们细说下这两个文件。

代码 C_with_S_c.c 的第 1 行是声明外部函数 asm_print，通知编译器这个函数并不在当前文件中定义。我们知道它定义在文件 C_with_S_S.S 中，但编译器是不知道的，所以只能在链接阶段将此函数重新定位，编排地址。

第 2～6 行是 c_print 的实现，在它的内部，它又调用了汇编代码 C_with_S_S.S 中的函数 asm_print，经过第 1 行的声明，我们要给它提供两个参数：字符串的起始地址及长度。

特别强调一下，由于这里并不打算把 C 标准库也链接进来，所以在求字符串长度时，我们不能用 string.h 中的 strlen 函数。也就是说即使 include <string.h>将其 strlen 的声明加进来，没有 strlen 实现本身所在的.o 文件也是不行的。函数声明的作用是：一方面告诉编译器该函数的参数所需要的栈空间大小及返回值，这样编译器能为其准备好执行环境；另一方面如果该函数是在外部文件中定义的，一定要在链接阶段时将其对应的目标文件一块链接进来。所以这里第 3～4 行通过 while 循环求字符串的长度。字符串结尾必须是空字符'\0'才行，否则 while 就是死循环了。这个字符串是代码 C_with_S_S.S 提供的，我们转过去看看。

在代码 C_with_S_S.S 的第 2 行，定义待打印的字符串时，在结尾人为地加了个 0，它就是空字符'\0'的 ASCII 码。

第 8～9 行是将_start 导出为全局符号，为的是给链接器用的，之前解释过了。

为了在汇编文件中引用外部的函数（未必是 C 代码中的），需要用 extern 来声明所需要的函数名。

由于我们用到了外部的函数 c_print，所以在第 7 行用 extern c_print 来声明 c_print 函数是外部的。第 11～13 行是为外部函数 c_print 压栈传参，调用它后清理栈空间。

第 16～17 行是调用 exit 系统调用告诉 Linux 我要正常退出。

在汇编语言中导出符号名用 global 关键字，这在之前说_start 时大伙已有所耳闻，global 将符号导出为全局属性，对程序中的所有文件可见，这样其他外部文件中也可以引用被 global 导出的符号啦，无论该符号是函数，还是变量。

由于在 c 代码文件 C_with_S_c.c 中也调用了这里的 asm_print 函数，所以为了让外部代码可以引用 asm_print。我们在第 19 行用 global asm_print 将其导出。

第 20 行之后是 asm_print 的实现，相信大家已经非常明白了，不解释。

通过这两个例子我相信大家已经掌握 C 和汇编混合编程的方法啦，确切说是方法之一。对于这种汇编代码和 C 代码单独编译的方式还是较容易的。

有关混合编程的部分就说完了，总结一下。

- 在汇编代码中导出符号供外部引用是用的关键字 global，引用外部文件的符号是用的关键字 extern。
- 在 C 代码中只要将符号定义为全局便可以被外部引用（一般情况下无需用额外关键字修饰，具体请参考 C 语言手册），引用外部符号时用 extern 声明即可。

6.3 实现自己的打印函数

一直以来，我们在往屏幕上输出文本时，要么利用 BIOS 中断，要么利用系统调用，这些都是依赖别人的方法。咱们还用过一个稍微有点独立的方法，就是直接写显存，但这貌似又没什么技术含量。如今我们要写一个打印函数了。

6.3.1 显卡的端口控制

在第 3 章我们讲述了有关显卡的知识，但当时怕影响兄弟们的学习积极性，我们并没有说把有关显卡的寄存器罗列出来。如今我们需要通过端口来控制显卡的行为，有些问题还是要面对的。

之前咱们对显卡的操作和对普通内存操作是一样的，打印字符时，就是往显存中 mov 一些字符的 ASCII 码和属性，那还是我们在显存默认的文本模式下。您想，我们都爱看视频、电影，话说十年前第一次看到 DVD 版本的电影时我都被震撼到了，当时看的是《星河战队》，清晰到毛发可见的程度，何况大家现在都偏爱蓝光高清版本，总之能够让我们看到如此炫丽的画面，这都是显卡的功劳，这说明显卡还可以工作在彩色图形模式。对于显卡的操作可不是咱们之前的 mov 来 mov 去就行了。不过我们也并不需要

那么复杂的功能，咱们还是在 80*25 的文本模式下转悠，而且还只是简单的操作。

之前我们已经对硬盘有过端口操作了，无非就是用 in 和 out 指令加不同的端口号，对显卡也是如此。显卡中的寄存器很多，不，是非常多，这里按照它们在图形管线（位于 CPU 和 video 之间）中的位置的顺序给大家介绍下，见表 6-2。

表 6-2　VGA 寄存器

寄存器分组	寄存器子类	读端口	写端口	说　明
Graphics Registers	Address Register	3CEh		
	Data Register	3CFh		
Sequencer Registers	Address Register	3C4h		
	Data Register	3C5h		
Attribute Controller Registers	Address Register	3C0h		
	Data Register	3C1h	3C0h	
CRT Controller Registers	Address Register	3x4h		x 的值取决于 Input/Output Address Select 字段，它决定映射的端口号为 3B4h-3B5h 或 3D4h-3D5h
	Data Register	3x5h		
Color Registers	DAC Address Write Mode Register	3C8h		
	DAC Address Read Mode Register	不可用	3C7h	
	DAC Data Register	3C9h		
	DAC State Register	3C7h	不可用	
External（General）Registers	Miscellaneous Output Register	3CCh	3C2h	
	Feature Control Register	3CAh	3xAh	写端口为 3BAh（mono 模式）或 3DAh（color 模式）
	Input Status #0 Register	3C2h	不可用	Read-only at 3C2h
	Input Status #1 Register	3xAh	不可用	读端口为 3BAh（mono 模式）或 3DAh（color 模式）

如您所见，表 6-2 中列出的寄存器的数量似乎没我说的那么恐怖，其实这些只是寄存器的目录而已。

以上所说的目录其实就是寄存器分组，在这些寄存器中也不全是分组。前四组寄存器属于分组，它们有一个特征，就是被分成了两类寄存器，即 Address Register 和 Data Register。这两个寄存器是干吗的呢？这得先从寄存器为什么要分成组开始说。

端口实际上就是 IO 接口电路上的寄存器，为了能访问到这些 CPU 外部的寄存器，计算机系统为这些寄存器统一编址，一个寄存器被赋予一个地址，这些地址可不是我们所说的内存地址，内存地址是用来访问内存用的，其范围取决于地址总线的宽度，而寄存器的地址范围是 0～65535（Intel 系统）。这些地址就是我们所说的端口号，用专门的 IO 指令 in 和 out 来读写这些寄存器。至于计算机内部访问端口怎么实现的，这是硬件工程师的事，咱们暂且奉行拿来主义，认同这个事实就够了。

IO 接口电路上的寄存器数量有多有少，这要看具体的外设了，我这么说您就明白了，这里给寄存器分组的原因是显卡（显示器的 IO 接口电路）上的寄存器太多了，如果一个寄存器就要占用一个系统端口的话，这得多浪费硬件资源，万一别的硬件也这么干，这 63336 个地址可就捉襟见肘了。所以计算机系统说了，我不管你们内部有多少寄存器，给你们的端口地址是有数的，你们自己内部协调吧。

计算机工程师是非常聪明的，把数据结构中数组的知识用到了硬件中。他们把每一个寄存器分组视为一个寄存器数组，提供一个寄存器用于指定数组下标，再提供一个寄存器用于对索引所指向的数组元素（也就是寄存器）进行输入输出操作。这样用这两个寄存器就能够定位寄存器数组中的任何寄存器啦。

这两个寄存器就是各组中的 Address Register 和 Data Register。Address Register 作为数组的索引（下标），Data Register 作为寄存器数组中该索引对应的寄存器，它相当于所对应的寄存器的窗口，往此窗口读写的数据都作用在索引所对应的寄存器上。

所以，对这类分组的寄存器操作方法是先在 Address Register 中指定寄存器的索引值，用来确定所操作的寄存器是哪个，然后在 Data Register 寄存器中对所索引的寄存器进行读写操作。

上面 CRT Controller Registers 寄存器组中的 Address Register 和 Data Register 的端口地址有些特殊，它的端口地址并不固定，具体值取决于 Miscellaneous Output Register 寄存器中的 Input/Output Address Select 字段，现在咱们看一下这个寄存器，见表 6-3。

表 6-3　Miscellaneous Output Register（读端口 3CCh，写端口 3C2h）

7	6	5	4	3	2	1	0
VSYNCP	HSYNCP	O/E Page		Clock Select		RAM En.	I/OAS

和大家坦白一点，显卡参数还需要专业人士来解释，由于咱们用不到这么高深的设置，加之我对显卡没有深入学习，所以这里面有好多参数术语，我不敢随意翻译成中文，担心误导大家，所以我直接把此寄存器各字段的英文描述搬过来了，至于中文的意思，大家仁者见仁、智者见智吧，请您见谅，见表 6-4。

表 6-4　Miscellaneous Output Register 各字段英文描述

字　段	描　述
VSYNCP (Vertical Sync Polarity)	"Determines the polarity of the vertical sync pulse and can be used (with HSP) to control the vertical size of the display by utilizing the autosynchronization feature of VGA displays. = 0 selects a positive vertical retrace sync pulse."
HSYNCP (Horizontal Sync Polarity)	"Determines the polarity of the horizontal sync pulse. = 0 selects a positive horizontal retrace sync pulse."
O/E Page (Odd/Even Page Select)	"Selects the upper/lower 64K page of memory when the system is in an eve/odd mode (modes 0,1,2,3,7). = 0 selects the low page = 1 selects the high page"
Clock Select	This field controls the selection of the dot clocks used in driving the display timing. The standard hardware has 2 clocks available to it, nominally 25 Mhz and 28 Mhz. It is possible that there may be other "external" clocks that can be selected by programming this register with the undefined values. The possible value of this register are: 00 -- select 25 Mhz clock (used for 320/640 pixel wide modes) 01 -- select 28 Mhz clock (used for 360/720 pixel wide modes) 10 -- undefined (possible external clock) 11 -- undefined (possible external clock)
RAM En. (RAM Enable)	"Controls system access to the display buffer. = 0 disables address decode for the display buffer from the system = 1 enables address decode for the display buffer from the system"
I/OAS (Input/Output Address Select)	"This bit selects the CRT controller addresses. When set to 0, this bit sets the CRT controller addresses to 0x03Bx and the address for the Input Status Register 1 to 0x03BA for compatibility with the monochrome adapter.　When set to 1, this bit sets CRT controller addresses to 0x03Dx and the Input Status Register 1 address to 0x03DA for compatibility with the color/graphics adapter. The Write addresses to the Feature Control register are affected in the same manner."

这里 I/OAS（Input/Output Address Select）字段不仅影响 CRT Controller Registers 寄存器组的 Address Register 和 Data Register 的端口地址，而且还影响 Feature Control register 寄存器的写端口地址和 Input Status #1 Register 寄存器的端口地址（此寄存器只有读端口），也就是影响了表 6-2 中所有端口地址中包括 x 的寄存器。所以，此字段意义重大，尽管咱们用不着设置（后面解释），我还是斗胆给大家翻译了，希望别误导大家，大概意思如下。

- I/OAS（Input/Output Address Select）

此位用来选择 CRT controller 寄存器组的地址，这里是指 Address Register 和 Data Register 的地址。

- 当此位为 0 时：

CRT controller 寄存器组的端口地址被设置为 0x3Bx，结合表 6-2，Address Register 和 Data Register 的端口地址实际值为 3B4h-3B5h。并且为了兼容 monochrome 适配器（显卡），Input Status #1 Register 寄存器的端口地址被设置为 0x3BA。

- 当此位为 1 时：

CRT controller 寄存器组的端口地址被设置为 0x3Dx，结合表 6-2，Address Register 和 Data Register 的端口地址实际值为 3D4h-3D5h。并且为了兼容 color/graphics 适配器（显卡），Input Status #1Register 寄存器的端口地址被设置为 0x3DA。

Feature Control register 寄存器的写端口也是 3xAh 的形式，该端口地址取值以同样的方式受 I/OAS 位的影响。

- 如果 I/OAS 位为 0，写端口地址为 3BAh。
- 如果 I/OAS 位为 1，写端口地址为 3DAh。

翻译结束，对以上中文版请大家自行斟酌。

默认情况下，Miscellaneous Output Register 寄存器的值为 0x67，其他字段不管，咱们只关注这最重要的 I/OAS 位，其值为 1。也就是说：

- CRT controller 寄存器组的 Address Register 的端口地址为 0x3D4，

Data Register 的端口地址 0x3D5。

- Input Status #1Register 寄存器的端口地址被设置为 0x3DA。
- Feature Control register 寄存器的写端口是 0x3DA。

坦白说，由于咱们涉及到的显卡操作只用到了 CRT Controller Registers 分组中的寄存器，其他分组中的寄存器咱们没用到，目前掌握的内容已经可以让我们继续后面的任务了。虽然如此，我还是将其余分组的寄存器都列出来了，万一有同学对其他分组寄存器感兴趣也用得着，见表 6-5～表 6-8。

表 6-5　CRT Controller Data Registers

索　引	寄　存　器	索　引	寄　存　器
00h	Horizontal Total Register	0Dh	Start Address Low Register
01h	End Horizontal Display Register	0Eh	Cursor Location High Register
02h	Start Horizontal Blanking Register	0Fh	Cursor Location Low Register
03h	End Horizontal Blanking Register	10h	Vertical Retrace Start Register
04h	Start Horizontal Retrace Register	11h	Vertical Retrace End Register
05h	End Horizontal Retrace Register	12h	Vertical Display End Register
06h	Vertical Total Register	13h	Offset Register
07h	Overflow Register	14h	Underline Location Register
08h	Preset Row Scan Register	15h	Start Vertical Blanking Register
09h	Maximum Scan Line Register	16h	End Vertical Blanking
0Ah	Cursor Start Register	17h	CRTC Mode Control Register
0Bh	Cursor End Register	18h	Line Compare Register
0Ch	Start Address High Register		

表 6-6　Graphics Registers

索　引	寄　存　器
00	Set/Reset Register
01	Enable Set/Reset Register
02	Color Compare Register
03	Data Rotate Register
04	Read Map Select Register
05	Graphics Mode Register
06	Miscellaneous Graphics Register
07	Color Don't Care Register
08	Bit Mask Register

表 6-7 Sequencer Registers

索　　引	寄　存　器
00	Reset Register
01	Clocking Mode Register
02	Map Mask Register
03	Character Map Select Register
04	Sequencer Memory Mode Register

表 6-8 Attribute Controller Registers

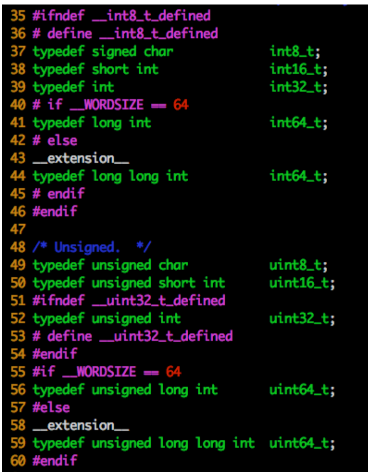
索　　引	寄　存　器
00-0Fh	Palette Registers（16 个寄存器）
10h	Attribute Mode Control Register
11h	Overscan Color Register
12h	Color Plane Enable Register
13h	Horizontal Pixel Panning Register
14h	Color Select Register

好啦，有关显卡寄存器的部分咱们就说完了，下面进入实习阶段。

6.3.2 实现单个字符打印

万事开头难，我们先从简单的打印字符开始。这个功能类似 C 语言中的 putchar，每次只打印一个字符，由于此函数咱们是在内核中实现的，暂且将其命名为 put_char。

在这之前，为了开发方便，我们定义一些数据类型。主要是参考了 Linux 的/usr/include/stdint.h 文件，有环境的同学可以自行看下，没环境的同学，请看图 6-7。



▲图 6-7 Linux 头文件 stdint.h 截图

该文件在我目前的 Linux 版本上是 320 行，这里只是冰山一角，里面各种宏显得好高大上啊，不过请放心，把这个图贴出来就是为了“吓唬”大家的，咱们不会写这么复杂，不信请看代码 6-1。

代码 6-1 （project/c6/a/lib/stdint.h）

```
 1 #ifndef __LIB_STDINT_H
 2 #define __LIB_STDINT_H
 3 typedef signed char int8_t;
 4 typedef signed short int int16_t;
 5 typedef signed int int32_t;
 6 typedef signed long long int int64_t;
 7 typedef unsigned char uint8_t;
 8 typedef unsigned short int uint16_t;
 9 typedef unsigned int uint32_t;
10 typedef unsigned long long int uint64_t;
11 #endif
```

怎么样，确实是很简单吧。以后我们采用的任何数据类型就要用这些定义好的啦。估计大家也注意到啦，咱们定义的 stdint.h 文件位于 lib 目录下，也就是说我新建了个 lib 目录用来专门存放各种库文件。不仅如此，在 lib 目录下还建立了 user 和 kernel 两个子目录，以后供内核使用的库文件就放在 lib/kernel/下，lib/user/中是用户进程使用的库文件。

我们要实现的字符打印函数叫 put_char，它是用汇编语言写的。因为要和显卡打交道啦，里面涉及到端口的读写操作，目前还是用纯汇编文件较方便，以后慢慢发展起来后，咱们会采取内联汇编的方式。

直接上代码啦，我们的打印函数统统在 print.S 文件中完成，该文件是各种打印函数的核心，重中之

重，这里先给大家介绍下它的处理流程。

（1）备份寄存器现场。

（2）获取光标坐标值，光标坐标值是下一个可打印字符的位置。

（3）获取待打印的字符。

（4）判断字符是否为控制字符，若是回车符、换行符、退格符三种控制字符之一，则进入相应的处理流程。否则，其余字符都被粗暴地认为是可见字符，进入输出流程处理。

（5）判断是否需要滚屏。

（6）更新光标坐标值，使其指向下一个打印字符的位置。

（7）恢复寄存器现场，退出。

该文件相对来说又有点长，故需要将其拆分成 3 部分，先给大伙儿呈上其第一部分，见代码 6-2-1。

代码 6-2-1　（project/c6/a/lib/kernel/print.S）

```
 1 TI_GDT equ  0
 2 RPL0  equ   0
 3 SELECTOR_VIDEO equ (0x0003<<3) + TI_GDT + RPL0
 4
 5 [bits 32]
 6 section .text
 7 ;-----------------------   put_char   ----------------------------
 8 ;功能描述：把栈中的 1 个字符写入光标所在处
 9 ;-------------------------------------------------------------------
10 global put_char
11 put_char:
12     pushad                    ;备份 32 位寄存器环境
13     ;需要保证 gs 中为正确的视频段选择子
       ;为保险起见，每次打印时都为 gs 赋值
14    mov ax, SELECTOR_VIDEO    ; 不能直接把立即数送入段寄存器
15     mov gs, ax
16
17 ;;;;;;;;;  获取当前光标位置 ;;;;;;;;;
18     ;先获得高 8 位
19     mov dx, 0x03d4            ;索引寄存器
20     mov al, 0x0e              ;用于提供光标位置的高 8 位
21     out dx, al
22     mov dx, 0x03d5            ;通过读写数据端口 0x3d5 来获得或设置光标位置
23     in al, dx                 ;得到了光标位置的高 8 位
24     mov ah, al
25
26     ;再获取低 8 位
27     mov dx, 0x03d4
28     mov al, 0x0f
29     out dx, al
30     mov dx, 0x03d5
31     in al, dx
32
33     ;将光标存入 bx
34     mov bx, ax
35     ;下面这行是在栈中获取待打印的字符
36     mov ecx, [esp + 36]       ;pushad 压入 4×8＝32 字节，
                                 ;加上主调函数 4 字节的返回地址，故 esp+36 字节
37     cmp cl, 0xd               ;CR 是 0x0d，LF 是 0x0a
38     jz .is_carriage_return
39     cmp cl, 0xa
40     jz .is_line_feed
41
42     cmp cl, 0x8               ;BS(backspace)的 asc 码是 8
43     jz .is_backspace
44     jmp .put_other
45 ;;;;;;;;;;;;;;;;;;
```

put_char 函数是以后我们任何一个打印功能的核心，所以光它的实现就要 112 行，这似乎是我们目前写过的最长的一个函数了，我保证以后也没有这么长的啦。好啦，长归长，不过也没什么难度，下面咱们开讲啦。

put_char 的打印原理是直接写显存，在 32 位保护模式下对内存的操作是“[段基址（选择子）：段内偏

移量]”，所以这就涉及视频段选择子啦。一直以来我们都是用段寄存器 gs 来存储视频段选择子的，以后也是，所以得保证在写显存之前，gs 中的值是正确的选择子。第 14～15 行是我们为 GS 寄存器赋值的代码，别小看这两行，大有来头，可不亚于摊上大事呢，吼吼，待咱们把 put_char 函数说完再跟大家好好说道说道吧，大家要做好心理准备。咱们先说别的。

第 1～3 行是定义了视频段的选择子，由于只需要这三行，专门定义个配置文件有点不值当的，所以直接在这定义了，好的习惯是放在配置文件中，大家在实践中不要学我。

第 10 行是通过关键字 global 把函数 put_char 导出为全局符号，这样对外部文件便可见了，外部文件通过声明便可以调用。

第 11 行开始定义函数 put_char。

第 12 行是用 pushad 指令备份 32 位寄存器的环境，按理说用到哪些寄存器就要备份哪些，我这里是偷懒行为，将 8 个 32 位全部备份了。PUSHAD 是 push all double，该指令压入所有双字长的寄存器，这里的“所有”一共是 8 个，它们的入栈先后顺序是：EAX->ECX->EDX->EBX-> ESP-> EBP->ESI->EDI，EAX 是最先入栈。

第 14～15 行是为 gs 安装正确的选择子，原因如前所述完事再说。

我们在打印字符时，通常都不用指定字符显示的坐标位置，大家也没觉得有什么奇怪，原因是字符是在当前光标的位置处显示的，而且光标的位置会一直更新顺延，我们的字符一直跟着光标走，似乎光标就是字符的导航一样，而我们已经习惯了跟随光标。我想大伙儿已经清楚了光标和字符的关系了，对，它们的关系就是没有任何关系。“光标在哪字符就在哪”，这是我们人为有意设置的，我们是在光标处打印字符。也就是说，我们也可以不在光标处打印字符，让光标和字符的位置分开。这一点在理论上就能证明，我们知道打印字符本质上就是把字符写入在显存中的某个地址处。在文本模式 80*25 下的显存可以显示 80*25=2000 个字符，每个字符占 2 字节，低字节是字符的 ASCII 码，高字节是前景色和背景色属性，所以在 4000 字节的显存空间中，只要起始地址为偶数的任意 2 字节我们都可以写入字符，您看，这哪里是光标能限制的。光标只是个亮点，用来吸引用户眼球的，它能够帮助咱们快速找到屏幕上的活跃位置，它本身与字符显示的位置没有关系。

话虽然这么说，但光标的作用已经被认同为当前可输入或显示字符的位置，字符在光标处显示，这已经成了字符打印的传统观念，所以在咱们的实现中也要传承复制这种观念。

光标是什么？不要感到奇怪。

我们 Linux 用户最熟悉了，就是屏幕上那一小白竖块，和文本软件中的小竖线是一回事，它们都是用来告诉用户当前文本输入点在哪里的。光标是字符的坐标，只不过该坐标不是二维的，而是一维的线性坐标，是屏幕上所有字符以 0 为起始的顺序。在默认的 80*25 模式下，每行 80 个字符共 25 行，屏幕上可以容纳 2000 个字符，故该坐标值的范围是 0～1999。第 0 行的所有字符坐标是 0～24，第 1 行的所有字符坐标是 25～49，以此类推，最后一行的所有字符是 1975～1999。由于一个字符占用 2 字节，所以光标乘以 2 后才是字符在显存中的地址。

光标的坐标位置是存放在光标坐标寄存器中的，当我们在屏幕上写入一个字符时，光标的坐标并不会自动+1，因为光标跟随字符并不是必要的，比如我们想删除文本中的某个字符时，咱们就可以把光标移动到该字符后面，再按下 delete 键，这样字符就被删除了，这就是光标与字符分离的应用之一。所以，光标位置并不会自动更新，因为光标坐标寄存器是可写的，如果需要的话，程序员可以自己来维护光标的坐标。

为了在光标处打印字符，咱们得先知道光标在哪，所以第一件事就是读取光标坐标寄存器，获取光标坐标值。

之前咱们介绍显卡上那么多的寄存器终于发挥用处了，我们看下表 6-5CRT Controller Data Registers 中索引为 0Eh 的 Cursor Location High Register 寄存器和索引为 0Fh 的 Cursor Location Low Register 寄存器，这两个寄存器都是 8 位长度，分别用来存储光标坐标的低 8 位和高 8 位地址。

访问 CRT controller 寄存器组的寄存器，需要先往端口地址为 0x3D4 的 Address Register 寄存器中写入寄存器的索引，再从端口地址为 0x3D5 的 Data Register 寄存器读、写数据。

在代码第 17～31 行用来获取光标值，先在第 19～21 行设置待操作的寄存器索引，我们先获取的是坐标的高 8 位，所以要将索引 0x0e 写入 Address Register 寄存器，其端口为 0x03d4。

确定了要操作的寄存器是 Cursor Location High Register 后，我们在第 22～24 行通过 Data Register 寄

存器，其端口是 0x3d5，将坐标读入到 al 寄存器，由于 al 中是坐标的高 8 位，所以第 24 行将其存储在 ah 寄存器。也许您心存疑惑，既然要把坐标的高 8 位存到寄存器 ah 中，为什么不把 in 指令中的 al 换成 ah，变成 in ah, dx？还多倒腾一次干吗？真的抱歉，对于 in 指令，如果源操作是 8 位寄存器，目的操作数必须是 al，如果源操作数是 16 位寄存器，目的操作数必须是 ax。

第 26～32 行用同样的方法获取到坐标的低 8 位，至此，寄存器 ax 中是光标完整的 16 位坐标值。

第 35 行是将光标值从 ax 寄存器中复制到 bx，这么做的原因是习惯用寄存器 bx 做基址寻址，还记得吗，在 16 位实模式下基址寄存器必须是 bx 或 bp，变址必须是寄存器 si 或 di。在 32 位保护模式下没必要这么做了，基址和变址寄存器可以是全部的 32 位的通用寄存器，就是刚才用 pushad 指令压入的那 8 个，忘了往上翻翻。以后的处理都要基于 bx 寄存器了，在此知道 bx 现在已经是光标坐标值就行了，它是下一个可打印字符的位置。

第 36 行是获取栈中压入的字符的 ASCII 码，也就是待打印的字符，这是 1 字节的数据。栈中除了调用 put_char 函数的返回地址占 4 字节外，还有最开始的 pushad 指令压入的 8 个 32 位的通用寄存器共 32 字节的数据，所以待打印的字符在栈顶偏移 36 字节的位置。

之后的第 36～44 行开始判断参数是什么字符，咱们这里只把回车符 CR（carriage_return）、换行符 LF（line_feed）和退格键 backspace 当作不可见字符，按照其实际控制意义来处理，其他字符暂时一律认为是可见字符。回车符的 ASCII 码是 0xd，换行符的 ASCII 码是 0xa，我们这里的处理是不管参数是回车符，还是换行符，一律按我们平时所理解的回车换行符（CRLF）处理（Linux 中就把换行符处理成回车+换行），即这两个动作的合成：光标回撤到行首+换到下一行。具体实现还得在代码 6-2-2 中处理回车换行符时再说，处理退格键也是在后面的代码中，咱们一会儿再说。

说到这，不知道是否让您有些意外，对于回车、换行等控制字符，其实际意义还需要咱们自己手动来处理？难道我直接写入回车的 ASCII 码 0xd 并不生效？答案让您失望了，必须是，咱们得手工展示控制字符的行为，说白了，咱们就像个控制字符的动作模拟器。硬件不负责这些字符编码的行为解释，字符编码的表现形式取决于软件的逻辑，所以，咱们甚至可以将 shift 键解释为换行，不过咱们还是按照常识来解释这些字符编码的行为，即换行符就是切换到下一行，退格键就是删除前一个字符。

代码 6-2-2　（project/c6/a/lib/kernel/print.S）

```
46
47  .is_backspace:
48 ;;;;;;;;;;;;      backspace 的一点说明       ;;;;;;;;;;
49 ; 当为 backspace 时，本质上只要将光标移向前一个显存位置即可.后面再输入的字符自然会覆盖此处的字符
50 ; 但有可能在键入 backspace 后并不再键入新的字符，这时光标已经向前移动到待删除的字符位置，但字符还在原处
51 ; 这就显得好怪异，所以此处添加了空格或空字符 0
52    dec bx
53    shl bx,1                    ;光标左移 1 位等于乘 2
                                  ;表示光标对应显存中的偏移字节
54    mov byte [gs:bx], 0x20      ;将待删除的字节补为 0 或空格皆可
55    inc bx
56    mov byte [gs:bx], 0x07
57    shr bx,1
58    jmp .set_cursor
59 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
60
61 .put_other:
62    shl bx, 1                   ;光标位置用 2 字节表示，将光标值乘 2
                                  ;表示对应显存中的偏移字节
63    mov [gs:bx], cl             ; ASCII 字符本身
64    inc bx
65    mov byte [gs:bx],0x07       ; 字符属性
66    shr bx, 1                   ; 恢复老的光标值
67    inc bx                      ; 下一个光标值
68    cmp bx, 2000
69    jl .set_cursor              ; 若光标值小于 2000，表示未写到
                                  ; 显存的最后，则去设置新的光标值
70                                ; 若超出屏幕字符数大小（2000）
                                  ; 则换行处理
71 .is_line_feed:                 ; 是换行符 LF(\n)
72 .is_carriage_return:           ; 是回车符 CR(\r)
73 ;如果是 CR(\r)，只要把光标移到行首就行了
```

```
74    xor dx, dx                      ; dx 是被除数的高 16 位，清 0
75    mov ax, bx                      ; ax 是被除数的低 16 位
76    mov si, 80                      ; 由于是效仿 Linux，Linux 中\n 表示
                                      ;下一行的行首，所以本系统中
77    div si                          ; 把\n 和\r 都处理为 Linux 中\n 的意思
                                      ;也就是下一行的行首
78    sub bx, dx                      ; 光标值减去除 80 的余数便是取整
79                                    ; 以上 4 行处理\r 的代码
80
81 .is_carriage_return_end:           ; 回车符 CR 处理结束
82    add bx, 80
83    cmp bx, 2000
84 .is_line_feed_end:                 ; 若是 LF(\n),将光标移+80 便可
85    jl .set_cursor
```

代码 6-2-2 处理控制键（不可见字符）：回车换行符及退格键，以及普通可见字符。在这之前，bx 已经在代码 6-2-1 第 34 行变成了下一个可打印字符的光标坐标值，光标值乘以 2 后便是光标在显存中的相对地址，接下来在该地址处写入字符。同样在代码 6-2-1 的第 36 行，寄存器 ecx 已经是待打印的参数了。由于字符的 ASCII 码只是 1 字节，所以只用寄存器 cl 就够了。

第 47～58 行是用来处理退格键 backspace 的代码。backspace 的原理就是将光标向回移动 1 位，将该处的字符用空格覆盖。第 52 行是用 dec 指令先将 bx 减 1，这样光标坐标便指向前一个字符了，第 53 行用 shl 指令将 bx 左移 1 位，相当于乘以 2，用 shl 指令做乘法比用 mul 指令方便。由于字符在显存中占 2 字节，低字节是 ASCII 码，高字符是属性，所以第 54 行在 bx 处，也就是低字节处先把空格的 ASCII 码 0x20 写入，第 55 行再通过 inc 指令把 bx 加上 1，这样 bx 便指向了该字符的属性位置，第 56 行再将属性 0x7 写入到高字节（其实您也看到了，不如直接在第 54 行写入 0x0720 最简单，循序渐进，后面我们会这么做）。0x7 表示黑屏白字，如果忘记了可以看下表 3-16。其实这是显卡默认的前景色和背景色，所以不加也行。这里显式地写入属性是为了提醒自己往显存中写字符，不仅要写 ASCII 码。此时的 bx 由于之前已经加 1 指向属性了，所以它现在已经变成了奇数，第 57 行通过右移指令 shr 将 bx 右移 1 位相当于除 2 取整，余数不要了，这样 bx 便由显存中的相对地址恢复成了光标坐标，此时的 bx 指向新覆盖的空格。在不考虑余数的情况下，用右移指令做除法比 div 指令要省事。由于删除了一个字符，bx 中的光标坐标已经被更新为前一位，之后在第 58 行跳到设置光标的流程.set_cursor，经过它的处理，光标才会显示在新位置。.set_cursor 的代码将在代码 6-2-3 中展示。

第 61～69 行处理可见字符的代码，其中第 62～66 行和上面处理 backspace 中的代码类似，区别是在第 67～68 行，通过 inc 指令把光标坐标 bx 值加 1，使 bx 成为新的可以打印字符的坐标，之后再判断这个新坐标是否超过了屏幕显示的范围，这个新坐标值就是下次打印字符的位置。前面说过了在 80*25 模式下屏幕可显示的字符数是 2000，这里用 cmp 指令把下次打印字符的坐标和 2000 比较，若小于 2000 则表示下次打印时，字符还会在当前屏幕的范围之内，于是在第 69 行直接跳转到.set_cursor 更新光标坐标。如果下次打印字符的坐标不小于 2000（在咱们的应用中顶多是等于 2000 的情况），这意味着需要滚屏了。

滚屏是什么？如您所料，就是滚动屏幕，让屏幕之外的内容可以显示在屏幕上，如图 6-8 所示。

以上我们所说的只是一种需要滚屏的情况，其实还有其他情况也需要滚屏。

（1）新的光标值超出了屏幕右下角最后一个字符的位置。

（2）最后一行中任意位置有回车或换行符。

咱们看一下实现滚屏的方法。



▲图 6-8　滚屏示意

之前咱们说过，在 80*25 文本模式下屏幕可显示 2000 个字（字符），4000 字节的内容。显存有 32KB，按理说显存中可以存放 32KB/4000B 约等于 8 屏的字符，这 8 屏的字符肯定不能一下子都显示在 1 个屏幕上，所以显卡就为咱们提供了两个寄存器，用来设置显存中那些在屏幕上显示的字符的起始位置，它们分别是表 6-5 中索引为 0xc 的 Start Address High Register 和 0xd 的 Start Address Low Register，这两个都是 8 位寄存器，如名字所示，它们分别设置地址的高 8 位和低 8 位。只要指定起始地址，屏幕自动从该地址起，向后显示 2000 个字符。注意，如果起始地址过大，显卡会将其在

显存中回绕 wrap around。好了，您已经猜到了，一种方案是通过设置显示的起始地址，让屏幕在显存中滑动起来实现窗口滚动。

其实，咱们还有另外一个方案。

默认情况下这两个寄存器的值是 0，也就是说默认情况下屏幕上的内容是从显存的首地址（物理地址）0xb8000 起，一直到以该地址向上偏移 3999 字节的地方。假如我们把屏幕固定在此，这样就不用再设置显示的起始地址了，其永远为 0。这两个寄存器 Start Address High Register 和 Start Address Low Register 我们就不用动了。

第一种方案的优势是显存中可缓存 16KB 个字符，屏幕外的文本也能找回。缺点就是要设置那两个起始地址寄存器，编程复杂一点。

第二种方案的优势就是编程简单，缺点是只能缓存 2000 个字符，屏幕上的字符便是全部缓存了。

话说，虽然第一种方案能缓存那么多个字符，但显存容量毕竟是有限的，当输出字节量超过 32KB 时，咱们照样要丢弃之前旧的字符，不是所有屏幕外的字符都能找回，而且要多个设置寄存器的操作。毕竟越简单越容易帮助大家看清操作系统原理，要不咱们就简单点，采取第二种方案。

屏幕每行 80 个字符，共 25 行，咱们的滚屏实现比较简单，现在说一下用此方案实现滚屏的步骤。

（1）将第 1～24 行的内容整块搬到第 0～23 行，也就是把第 0 行的数据覆盖。

（2）再将第 24 行，也就是最后一行的字符用空格覆盖，这样它看上去是一个新的空行。

（3）把光标移到第 24 行也就是最后一行行首。

经过这三步，屏幕就像向上滚动了一行（似乎我们是字符的搬运工）。另外，由于第 3 步只是计算好了新的坐标值并没有在光标寄存器中更新，所以它的顺序不是很重要，也可以放在前面先做。

回来继续说代码 6-2-2，第 71～84 行是处理回车换行符的代码，我们刚说的滚屏操作也需要它。因为在滚屏操作中，除了将当前屏幕所有内容上移一行外，光标的坐标也要更新为下一行的行首，这实际上相当于在最后一行的行尾键入了回车符，在此用处理回车换行符的代码帮助我们实现上面第 3 步的更新光标值。

回车换行本质上是两个操作，一个是回车 carriage_return，将光标回撤到当前行的行首，另一个是换行 line_feed，就是切换到下一行。这两个动作合成到一起便是我们平时敲下一个回车键的效果：光标出现在下一行行首。

第 74～78 行是在处理回车符，也就是将光标回撤到行首。这里的方法是将光标坐标值 bx 对 80 求模，再用坐标值 bx 减去余数就是行首字符的位置。第 75～77 行就是对 80 求模，经过 div 除法操作后，dx 寄存器中为余数。在第 78 行用坐标值减余数，经过“sub bx, dx”后，bx 便为当前行首坐标，实现了回车符的功能（不过目前还没有更新光标坐标寄存器，更新之后才算真正完成）。

接下来是处理换行符 line_feed，也就是将光标切换到下一行。方法是将当前光标坐标值加上每行的字符数 80，这样便是下一行的坐标啦。这是在第 82 行完成的，至此我们完成了回车、换行两个字符的处理，我们滚屏操作中的第 3 步也完成了。

第 83 行是回车换行符处理流程中自己的滚屏判断，它和前面所说的滚屏流程无关，不要误以为是接上面的滚屏说的。倘若没有之前的滚屏，在单独的回车换行处理流程中也要判断在将光标更新为下一行后是否超出了屏幕范围而需要滚屏。这就是我们前面所说的第 2 种需要滚屏的情况，即在最后一行中的任意一个位置有回车或换行符都将导致滚屏。

显然，我们这里需要换行的原因是由于屏幕已经满了，当前光标坐标（不是下一个可打印位置的光标坐标）已经是在最后一行的最后一列，需要滚屏了。我前面说过啦，这种情况下滚屏“相当于”在屏幕右下角后面敲入一个回车键，尽管没有回车符或换行符，但它能使屏幕新起一行，这恰恰是滚屏需要的。所以，这就是借用回车换行符处理流程的原因，它能为滚屏增加个新的空行。

从处理这几个控制字符大家就能够看出，字符集其实就是一套字符行为表现的约定，基本上可见字符表现在图形，不可见字符表现在动作。但这也只是约定，您懂的，将字符编码解释成什么样，完全取决于字符处理软件的意愿。

继续看代码 6-2-3，这是 print.S 的最后一部分，我们滚屏操作中的前 2 步和往光标寄存器中更新坐标

值都在这里。

代码 6-2-3 （project/c6/a/lib/kernel/print.S）

```
 86
 87 ;屏幕行范围是 0～24，滚屏的原理是将屏幕的第 1～24 行搬运到第 0～23 行
   ;再将第 24 行用空格填充
 88 .roll_screen:                ; 若超出屏幕大小，开始滚屏
 89    cld
 90    mov ecx, 960              ; 2000-80=1920 个字符要搬运，共 1920*2=3840 字节
                                 ;一次搬 4 字节，共 3840/4=960 次
 91    mov esi, 0xc00b80a0       ; 第 1 行行首
 92    mov edi, 0xc00b8000       ; 第 0 行行首
 93    rep movsd
 94
 95 ;;;;;;;将最后一行填充为空白
 96    mov ebx, 3840             ; 最后一行首字符的第一个字节偏移= 1920 * 2
 97    mov ecx, 80               ;一行是 80 字符（160 字节），每次清空 1 字符
                                 ; （2 字节），一行需要移动 80 次
 98 .cls:
 99    mov word [gs:ebx], 0x0720;0x0720 是黑底白字的空格键
100    add ebx, 2
101    loop .cls
102    mov bx,1920               ;将光标值重置为 1920，最后一行的首字符
103
104  .set_cursor:
105  ;将光标设为 bx 值
106 ;;;;;;; 1 先设置高 8 位 ;;;;;;;;
107     mov dx, 0x03d4           ;索引寄存器
108     mov al, 0x0e             ;用于提供光标位置的高 8 位
109     out dx, al
110     mov dx, 0x03d5           ;通过读写数据端口 0x3d5 来获得或设置光标位置
111     mov al, bh
112     out dx, al
113
114 ;;;;;;; 2 再设置低 8 位 ;;;;;;;;;
115     mov dx, 0x03d4
116     mov al, 0x0f
117     out dx, al
118     mov dx, 0x03d5
119     mov al, bl
120     out dx, al
121   .put_char_done:
122     popad
123     ret
```

本代码开头就是滚屏的部分.roll_screen，第 89 行先用 cld 指令清除方向位，就是把 eflags 寄存器中方向标志位 DF（Direction Flag）清 0。前面说过了，cld，字符串“搬运”指令 movs[bwd]和 rep 三剑客组合完成大数据的复制。

第 90 行是将 ecx 赋值为 960，它用来控制 rep 重复执行指令的次数，这里是要把第 1～24 行的字符整体往上提一行，复制到第 0～23 行。要复制的字符数是 2000-80=1920 个，每个字符是 2 字节，共 3840 字节。我们是用 movsd 指令来复制的，它一次复制 4 字节数据，所以需要执行 3840/4=960 次。

第 91 行是把要复制的起始地址赋给 esi 寄存器，也就是屏幕第 1 行的起始地址，物理地址是 0xb80a0。将来实现用户进程后，为方便用户进程的管理，此处会用其虚拟地址 0xc00b80a0 代替。

第 92 行是把目的地址赋给 edi 寄存器，也就是屏幕的第 0 行的起始地址，物理地址是 0xb8000，同上，将来也会用其虚拟地址 0xc00b8000 代替。

第 93 行用 rep 指令配合 movsd 指令，开始循环复制，直至把第 24 行的数据复制完毕。

滚屏操作还差一步，需要将最后一行用空格填充，这是在第 95～101 行中完成的。

第 96～97 行是在准备复制的起始地址和循环次数。最后一行在显存中的偏移地址是 3840，循环次数是每行的字节总数除以每次处理的字节数，具体循环次数要先看下面的实现部分。

第 98 行的 cls 是准备清空最后一行，第 99～101 行是循环处理屏幕最后一行中的每 1 个字符（2 字节），一次写入 2 字节数据将字符置为黑屏白字的空格符。0x0720 是一个空格的数据，低字节是空格的 ASCII 码 0x20，

高字节是前景色和背景色属性 0x07。这里通过“mov dword [gs：ebx]， 0x0720”处理 1 个空格，在第 101 行用了 loop 指令来实现循环执行，它也是用 ecx 作为循环计数器，每执行一次，ecx 自动减 1，直到为 0 时停止执行。

现在大伙儿知道为什么在第 97 行为 ecx 赋值为 80 了，每行 80 个字符共 160 字节，一次清空 1 个字符，即 2 字节，故循环次数是 160/2=80 次。

第 102 行的 mov bx，1920 是把 bx 设置为最后一行起始，bx 作为光标坐标值，进入.set_cursor 完成光标坐标的更新。

滚屏说完了，下面说一下光标设置。

第 104～120 行是.set_cursor，这是用来设置光标的代码，它把光标坐标寄存器设置为寄存器 bx 中的值。原理也是先通过 0x3d4 端口写入待操作寄存器的索引，这和之前获取坐标值的代码是一样的。只不过操作 0x3d5 端口不再是读取指令 in，而是写入指令 out，我们要把 bx 中的值更新到光标坐标寄存器的高 8 位和低 8 位。这部分代码我相信大家已经能看明白了，不再细说。

第 121～123 行完成了 put_char 的处理流程，用 popd 指令将之前入栈的 8 个 32 位寄存器恢复到各个的寄存器中，环境恢复，第 123 行 ret 指令返回。

其实我此时长呼一口气，终于说完了 put_char，不过，转念一想，似乎还有点小尾巴……我要说道说道之前在代码 6-2-1 中埋下的伏笔了，大伙儿再坚持一会儿。

下面这些内容是为了解释此处为 GS 赋值的原因，是为了避免将来摊上大事儿，这与特权级相关。先简要和大伙儿解释下，咱们还没有到真正使用特权级的地方，以后在说用户进程的时候才是它发挥的时机。引出特权级只是为了解释第 14～15 行为什么给 gs 赋值，它只是为了防止将来因为 GS 为 0 导致 CPU 抛异常才提前加的，这和 put_char 本身的功能是无关的。

好啦，进入正题。

和硬件相关的访问都属于内核的工作，打印输出也是，用户进程只能依靠内核的帮助来完成和硬件相关的操作，不过，这只是咱们的愿景，可不能保证用户进程会乖乖听话，得有一套机制来防止用户进程直接访问内核资源的这种越权行为。

检测这些“越权”的行为是由 CPU 负责的，不过不要觉得 CPU 有多智能，它只会老老实实不知疲倦地执行指令，根本不知道目前运行的指令是属于内核的，还是属于用户进程的，所以 CPU 分不出来谁是谁，更谈不上由它来找出越权了。真正起检测作用的是人为给 CPU 设置的规则，CPU 按照这套规则办事就妥妥的了，这里所说的规则就是特权级，用户进程反不了天，主要就是因为有特权级管着呢。特权级分为 0、1、2、3 共 4 个等级，数字越小特权越大，在 Linux 中，内核工作在 0 级，用户进程工作在 3 级，特权级 1、2 都空着没用到，咱们这里也是学习 Linux 的做法，内核在 0 级特权，用户进程在 3 级特权。先给大家简要介绍几个术语，以后我们会详细说明。CPL，即当前特权级，也就是程序运行时所处的特权等级。RPL，即请求特权级，它是选择子中的 RPL 字段。还有就是段描述符中已经说过的 DPL，它表示段描述符所代表的内存区域的特权级。

在 loader 中，我们早已经将 gs 赋予成正确的选择子，程序进入保护模式之后就继承了 0 特权级，内核也在 0 特权级下工作。到现在为止这一切都很正常，可是在不久的将来，当我们有了用户进程之后，问题来了。

为了解释清楚，需要提前和大家说一下，用户进程是需要用 iretd 返回指令上 CPU 运行的，这些以后会说的，大家暂且接受这个结论。CPU 在执行 iretd 指令时会做特权检查，它检查 DS、ES、FS、GS“数据”段寄存器的内容，这里的数据我加了引号，它只是修饰段寄存器的定语，用来指除代码段寄存器 CS 和栈段寄存器 SS 之外的段寄存器。在 32 位保护模式下它们中存储的都是选择子，要是有任何一个段寄存器所指向的段描述符的 DPL 权限高于从 iretd 命令返回后的 CPL（这个返回后的 CPL 也就是新的 CPL，CPL 就是加载到 CS 寄存器中选择子的 RPL），CPU 就会将该段寄存器赋值为 0。这样做是有意设计的，相当于一种间接的保护策略，它是如何起到保护作用的呢？没有迫害就谈不上保护，其实 CPU 这么做的原因就是怕高特权级的资源能被低特权级程序访问，能访问说不定就能破坏，所以 CPU 是不会让低特权级程序有访问高特权级资源的机会的。虽然已经将段寄存器置 0 了，但如果不访问的话，一切相安无事，一旦将来用该段寄存器访问内存时，由于选择子为 0，这表示选择子中的索引位、TI 位和 RPL 位都为 0，所以会在 GDT 中检索到第 0 个段描述符，由于第 0 个段描述符是空的不可用（这是 Intel 有意为之的，为避免忘记初始化选择子而误选到了 GDT 中的第 0 个描述符），这就会导致 CPU

抛出异常。总之，如果 CPU 发现新的 CPL 权限比数据段寄存器（DS、ES、FS、GS）指向的段描述符的 DPL 权限小，CPU 便认为这是一种越权访问。CPU 中的检测规则是人定的，人的思维是：访问别人，至少权限得和人家一样才行，所以计算机世界里的规则按照人类思维来理解是非常合理的。在这里可以理解为 CPU 发现用户进程想直接访问内核的资源，CPU 是绝对不能答应的，所以把权限不符合条件的寄存器填充为 0，给它点颜色看看。尽管选择子已经为 0，但如果用户进程乖乖地待在自己的地盘不去访问内核的空间，CPU 也不会难为它，用户进程就像个正常的程序一样，否则进程不老实的话，CPU 这才抛出异常呢。

好啦，看看我们实际的情况，用户进程的特权级由 CS 寄存器中选择子的 RPL 字段决定，它将成为进程在 CPU 上运行时的 CPL。将来为用户进程初始化寄存器时，CS 中的选择子 RPL 字段必须为 3，进而它就是从 iretd 指令返回后的新的 CPL。而我们用于访问显存的 GS 寄存器，在新的 CPL 为 3 的情况下，无论为其赋为何值，其选择子所指向的段描述符中的 DPL 值必须等于 3，否则 CPU 会将 GS 置为 0。我们目前使用的显存段描述符是全局描述符表 GDT 中的第 3 个段描述符，但它的 DPL 为 0，不为 3。您看，问题来了，怎么办呢？方法总比问题多，这里给出了两个方案。

（1）为用户进程创建一个显存段描述符，其 DPL 为 3，专门给特权 3 级的用户进程使用。

（2）在打印函数中动动手脚，将 GS 的值改为指向目前 DPL 为 0 的显存段描述符。

貌似第 1 个方案更合理，而且工作量也不大，但别忘记了，打印字符这种和硬件相关的功能属于内核的范畴，用户需要打印输出时，它应该请求内核的服务，由内核帮助完成，这才是我们的理想模式，绝对不能让用户进程直接操作显卡，否则要我大内核威严何在，用户进程得不到限制可就无法无天了。所以我们放弃第 1 种做法，干脆在初始化用户进程寄存器时，直接将 GS 赋值为 0。用户进程将来在打印输出时，是需要通过系统调用陷入内核来完成的，到时用户进程的 CPL 会由 3 变成 0，执行的是内核的代码，那时再将 GS 赋值为内核使用的显存段选择子即可。

所以，我们必须得把为 GS 赋值的代码放在打印时必须要调用的函数中，这就是咱们的 put_char 函数，在咱们的代码中，它是各种打印的核心。

顺便说下，陷入内核后的用户进程，由于其 GS 已经被咱们置为内核视频段描述符的选择子，其指向的 DPL 依然为 0，故从内核态返回后，GS 又会被 CPU 置为 0，不过这是后话了。

原因说完了，大伙儿现在了解在 put_char 函数中为 GS 赋值的原因了，不明白也没关系，知道第 14～15 行代码不是多余的即可，以后在实现用户进程时自然就明白了。

终于说完 put_char 了，我已经迫不及待地想试试了，咱们需要改进一下相关文件。

print.S 中的函数 put_char 对于其他文件来说属于外部函数，要是想在其他文件中用到的话，各文件得将其声明加进来。凡是用到此函数的文件都要加上其声明，这样比较麻烦，所以咱们还是将其写成头文件，谁需要它就将其包含进来就行了。请见代码 6-3。

代码 6-3 （project/c6/a/lib/kernel/print.h）

```
1 #ifndef __LIB_KERNEL_PRINT_H
2 #define __LIB_KERNEL_PRINT_H
3 #include "stdint.h"
4 void put_char(uint8_t char_asci);
5 #endif
6
```

为防止头文件被重复包含，避免头文件中的变量等出现重复定义的情况。可以用条件编译指令#ifndef 和#endif 来封闭文件的内容，把要定义的内容放在它们之中。

前 2 行是以 print.h 所在的路径定义了这个宏__LIB_KERNEL_PRINT_H，以该宏来判断是否重复包含。

第 3 行是通过 include 指令包含了“stdint.h”。这里是用双引号括住了 stdint.h，目的是包含自己指定的文件，如果是用尖括号<>括住的，这是让编译器到系统头文件所在的目录中找所包含的文件，这个目录通常是/usr/include。

第 4 行就是一句简单的声明，给出 put_char 函数的原型，您看，虽然 put_char 是用汇编语言写的，但它被 C 语言引用时，在 C 语言中的形式还是得符合 C 语言语法，加之咱们之前已经讲过了 cdecl 调用约定，

所以 put_char 的 C 语言形式是 void put_char（uint8_t char_asci），这样 put_char 才能从栈中获取参数 char_asci。这里的 char_asci 是无符号 8 位整型变量（其实就是 unsigned char），这与代码 6-2-1 第 36 行获取参数到寄存器 ecx 后，只用寄存器 cl 是相吻合的。

第 5 行#endif 是与条件编译#ifndef 相配合的结束指令，固定用法。

好啦，下面咱们改进 main.c，在其中用 put_char 函数打印字符。

代码 6-4　（project/c6/a/kernel/main.o）

```
 1 #include "print.h"
 2 void main(void) {
 3    put_char('k');
 4    put_char('e');
 5    put_char('r');
 6    put_char('n');
 7    put_char('e');
 8    put_char('l');
 9    put_char('\n');
10    put_char('1');
11    put_char('2');
12    put_char('\b');
13    put_char('3');
14    while(1);
15 }
```

由于咱们还没有实现打印字符串的函数，所以用了多个 put_char 函数来拼凑字符串。

代码前 8 行应该是打印字符串 kernel，第 9 行换行，第 10～11 行打印的是字符 12，但字符 2 马上被第 12 行的退格键 backspace 删除。第 13 行，在字符 1 的后面将输出字符 3。

好啦，还是得经过编译、链接、写入虚拟硬盘三步。

编译 print.S

nasm -f elf -o lib/kernel/print.o lib/kernel/print.S 回车

编译 main.c

gcc -I lib/kernel/ -c -o kernel/main.o kernel/main.c 回车

链接 main.o 和 print.o

ld -Ttext 0xc0001500 -e main -o kernel.bin \

> kernel/main.o lib/kernel/print.o 回车

注意，上面 ld 命令第一行最后的斜杠'\'不属于命令本身，用于一行写不下全部命令参数的情况。

写入虚拟硬盘

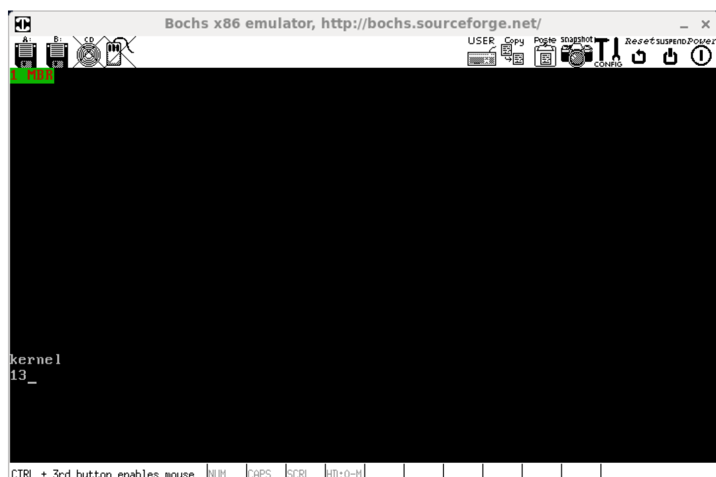
```
dd if=kernel.bin of=/home/work/my_workspace/bochs/hd60M.img \
bs=512 count=200 seek=9 conv=notrunc
```

好了，终于到了这一刻，这是我们第 1 个像模像样的打印函数，希望前面漫长的努力没有白费，我现在多少还有点紧张呢。兄弟们，上机运行看效果，如图 6-9 所示。

看，输出与我们预期的是一样的，在字符串 kernel 结尾的换行符起了作用，在下一行，原本是输出“123”，但'2'和'3'之间的退格键将字符'2'删除了，只留下了“13”。另外，由于当前光标坐标寄存器中有残余数值，我们并没有将其清 0，所以并未在屏幕左上角开始输出。

在上机运行之后，我这里提醒下大家链接文件时的顺序问题。在上面的链接阶段，目标文件链接顺序是 main.o 在前，print.o 在后。大家知道，main.c 文件中用到了 print.o 中的 put_char 函数，在链接顺序上，属于“调用在前，实现在后”的顺序。如果将 print.o 放在前面，main.o 放在后面，也就是实现在前，调用在后，此时生成的可执行文件起始虚拟地址并不准确，会有向后顺延的现象，并且 segment 的数量也不一样。原因是链接器对符号表的处理细节造成的，链接器主要工作就是整合目标文件中的符号，为其分配地址，让使用该符号的文件可以正确定位到它。由于我能力有限，只能大概说一下我的理解，链接器最先处理的目标文件是参数中从左边数第一个（咱们这里是 main.o），对于里面找不到的符号（这里是 put_char），链接器会将它记录下来，以备在后面的目标文件中查找。如果将其顺序颠倒，势必导致在后面的目标文件

中才出现找不到符号的情况，而该符号的实现所在的目标文件早已经过去了，这可能使链接器内部采用另外的处理流程，导致出来的可执行程序不太一样。



▲图 6-9　put_char 效果

所以，建议大家最好保持调用在前，实现在后的顺序来链接。

6.3.3　实现字符串打印

上一节的内容实在是太长了，不过本节内容就非常少了，在本节将实现一个字符串打印函数，它实际上就是对上一节单个字符打印函数的封装。

我们在 C 语言中定义的字符串，C 编译器会把字符串的结尾自动加上'\0'，用它作为字符串的结束标记，有了这个标记咱们才能确定字符串的长度。顺便说一句，'\0'的 ASCII 码为 0，所以很多字符串函数内部都通过把各个字符与 0 比较来判断字符串是否结束。

请见代码 6-5。

代码 6-5　（project/c6/b/lib/kernel/print.S）

```
 5 [bits 32]
 6 section .text
 7 ;-------------------------------------------------------------------
 8 ;put_str 通过 put_char 来打印以 0 字符结尾的字符串
 9 ;-------------------------------------------------------------------
10 ;输入:栈中参数为打印的字符串
11 ;输出:无
12
13 global put_str
14 put_str:
15 ;由于本函数中只用到了 ebx 和 ecx，只备份这两个寄存器
16     push ebx
17     push ecx
18     xor ecx, ecx            ; 准备用 ecx 存储参数，清空
19     mov ebx, [esp + 12]     ; 从栈中得到待打印的字符串地址
20 .goon:
21     mov cl, [ebx]
22     cmp cl, 0               ; 如果处理到了字符串尾，跳到结束处返回
23     jz .str_over
24     push ecx                ; 为 put_char 函数传递参数
25     call put_char
26     add esp, 4              ; 回收参数所占的栈空间
27     inc ebx                 ; 使 ebx 指向下一个字符
28     jmp .goon
29 .str_over:
30     pop ecx
31     pop ebx
32     ret
```

开门见山啦，put_str 函数是我们的字符串打印函数，它的原理是每次处理一个字符，循环调用 put_char 来完成字符串中全部字符的打印，所以代码并不长，咱们说清楚它是分分钟的事。

第 13 行导出 put_str 函数供其他外部文件调用。

第 14 行是 put_str 正式开始，由于在实现中只用到了 ebx 和 ecx 两个寄存器，所以这里在第 16～17 行直接将这两个寄存器入栈备份，并没有像 put_char 中那样偷懒，用 pushad 一次保存所有 32 位通用寄存器。

第 18～19 行中，由于下面要用寄存器 ecx 来传递字符，所以在此先将其用异或指令 xor 清 0。接下来要获取待打印的字符串了，大家要注意，C 编译器会为字符串常量分配一块内存，在这块内存中存储字符串中各字符的 ASCII 码，并且会在结尾后自动补个结束字符'\0'，它的 ASCII 码是 0。编译器将该字符串作为参数时，传递的是字符串所在的内存起始地址，也就是说压入到栈中的是存储该字符串的内存首地址。其实想想看这也是有道理的，如果是将字符串中各个字符的 ASCII 码都压入栈中，由于字符串长度不定，占用的栈空间也不定，有可能会造成栈溢出呢，而且也不知道到底有多少个字符，如何回收参数所占的栈空间也是问题，总之，直接把字符串中各字符当作参数是绝对不靠谱啊。好啦，回到正题，在咱们的 put_str 中，从栈中获取到的参数是字符串内存地址，我们需要对该地址进行内存寻址才能找到字符的 ASCII 码。前面在栈中备份了两个寄存器占 8 字节，再加上栈中 put_str 函数的返回地址占 4 字节，故参数在栈中偏移栈顶 12 字节的位置。

第 19 行将字符串首地址存储到 ebx，习惯用 ebx 作为基址寻址了。

第 20～28 行分别寻址每一个字符，分别调用 put_char 逐个打印。

在第 21 行中，通过 ebx 寻址访问内存获取 1 字节数据，存储到寄存器 cl 中，此时 cl 便是字符的 ASCII 码。

在第 22 行，将寄存器 cl 与 0 比较来判断字符串是否处理结束，如果比较的结果为 0，说明处理完毕，在第 23 行跳转到.str_over 结束返回。否则进入第 24 行，将 ecx 作为参数传递给 put_char。尽管 cl 中才是字符的 ASCII 码，但 32 位保护模式下的栈，要么压入 16 位操作数，要么压入 32 位操作数，而我比较习惯使用 32 位操作数，所以这里干脆把 ecx 整个压栈了。

第 26 行回收参数所占的 4 字节空间。

第 27 行使 ebx 加 1，让其指向字符串中的下一个字符。

好啦，如我刚才所说，函数确实不长，分分钟就说完了，似乎该是检测成果的时候了，别忘了，为了能在 main.c 中调用它还得修改相关文件，一个是在 print.h 中的增加 put_str 的函数声明，另一个是在 main.c 中调用它。

代码 6-6　（project/c6/b/lib/kernel/print.h）

```
1 #ifndef __LIB_KERNEL_PRINT_H
2 #define __LIB_KERNEL_PRINT_H
3 #include "stdint.h"
4 void put_char(uint8_t char_asci);
5 void put_str(char* message);
6 #endif
7
```

没什么好说的，就是在第 5 行加了个 put_str 的声明，前面说过了参数是地址，由于是字符串的地址，所以这里的类型是字符型指针 char*。

代码 6-7　（project/c6/b/kernel/main.c）

```
1 #include "print.h"
2 void main(void) {
3    put_str("I am kernel\n");
4    while(1);
5 }
```

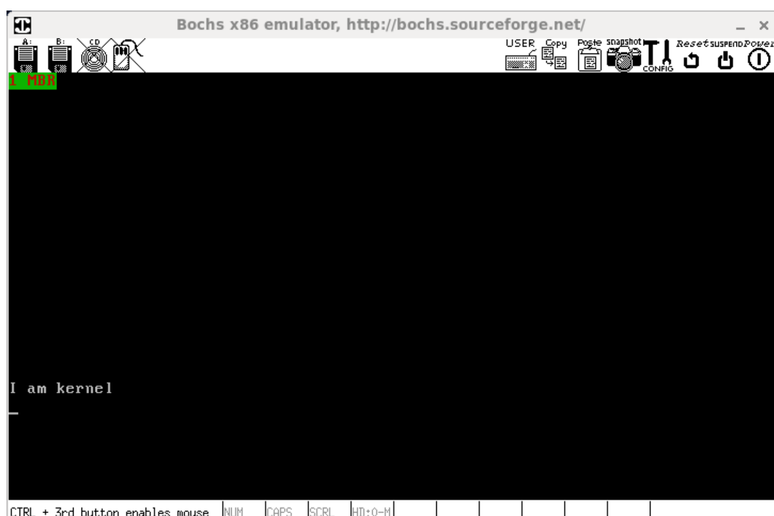
代码 6-7 的 main 函数中，把之前连续多个 put_char 调用去掉了，直接改成用 put_str 实现打印。大伙切记，在完成之前，while 这个死循环千万不要丢，否则 CPU 就“跑的没影了”。

编译链接同之前一样，直接看运行结果，如图 6-10 所示。

大家看到了屏幕上的输出字符串“I am kernel”，并且，换行符'\n'使光标到了下一行。结果符合预期，完成。

几下就完成了一个函数，看来前面在 put_char 函数上花费的时间是值得的。下一个要实现的函数

也很短。



▲图 6-10 put_str 运行效果

6.3.4 实现整数打印

前两节是完成了有关字符的打印，这一节中，我们要实现数字打印，当然仅仅是整数，咱们的实现中不支持浮点数。用于打印整数的函数名是 put_int，它的原理是将数字转换成对应的字符，比如数字 9 变成字符 '9'。这么一说，大家的潜意识里觉得依然是封装 put_char，是的没错，不过要麻烦一些，多说无益，别把大家说闹心了，请见代码 6-8。

代码 6-8 （project/c6/c/lib/kernel/print.S）

```
…略
5 section .data
6 put_int_buffer  dq  0          ; 定义 8 字节缓冲区用于数字到字符的转换
…略
155 ;----------将小端字节序的数字变成对应的 ASCII 后，倒置----------
156 ;输入:栈中参数为待打印的数字
157 ;输出:在屏幕上打印十六进制数字，并不会打印前缀 0x
    ;如打印十进制 15 时，只会直接打印 f，不会是 0xf
158 ;--------------------------------------------------------------------------------------
---
159 global put_int
160 put_int:
161     pushad
162     mov ebp, esp
163     mov eax, [ebp+4*9]      ; call 的返回地址占 4 字节+pushad 的 8 个 4 字节
164     mov edx, eax
165     mov edi, 7              ; 指定在 put_int_buffer 中初始的偏移量
166     mov ecx, 8              ; 32 位数字中，十六进制数字的位数是 8 个
167     mov ebx, put_int_buffer
168
169 ;将 32 位数字按照十六进制的形式从低位到高位逐个处理
    ;共处理 8 个十六进制数字
170 .16based_4bits:             ; 每 4 位二进制是十六进制数字的 1 位
 ;遍历每一位十六进制数字
171     and edx, 0x0000000F     ;解析十六进制数字的每一位
                            ; and 与操作后，edx 只有低 4 位有效
172     cmp edx, 9              ; 数字 0～9 和 a～f 需要分别处理成对应的字符
173     jg .is_A2F
174     add edx, '0'            ; ASCII 码是 8 位大小。add 求和操作后，edx 低 8 位有效
175     jmp .store
176 .is_A2F:
177     sub edx, 10             ; A～F 减去 10 所得到的差，再加上字符 A 的
                            ; ASCII 码，便是 A～F 对应的 ASCII 码
178     add edx, 'A'
```

```
179
180 ;将每一位数字转换成对应的字符后，按照类似“大端”的顺序
     ;存储到缓冲区 put_int_buffer
181 ;高位字符放在低地址，低位字符要放在高地址，这样和大端字节序
     ;类似,只不过咱们这里是字符序
182 .store:
183 ; 此时 dl 中应该是数字对应的字符的 ASCII 码
184     mov [ebx+edi], dl
185     dec edi
186     shr eax, 4
187     mov edx, eax
188     loop .16based_4bits
189
190 ;现在 put_int_buffer 中已全是字符，打印之前
191 ;把高位连续的字符去掉，比如把字符 000123 变成 123
192 .ready_to_print:
193     inc edi                ; 此时 edi 退减为-1(0xffffffff)，加 1 使其为 0
194 .skip_prefix_0:
195     cmp edi,8              ; 若已经比较第 9 个字符了
                           ; 表示待打印的字符串为全 0
196     je .full0
197 ;找出连续的 0 字符, edi 作为非 0 的最高位字符的偏移
198 .go_on_skip:
199     mov cl, [put_int_buffer+edi]
200     inc edi
201     cmp cl, '0'
202     je .skip_prefix_0      ; 继续判断下一位字符是否为字符 0（不是数字 0）
203     dec edi                ;edi 在上面的 inc 操作中指向了下一个字符
    ;若当前字符不为'0',要使 edi 减 1 恢复指向当前字符
204     jmp .put_each_num
205
206 .full0:
207     mov cl,'0'             ; 输入的数字为全 0 时，则只打印 0
208 .put_each_num:
209     push ecx               ; 此时 cl 中为可打印的字符
210     call put_char
211     add esp, 4
212     inc edi                ; 使 edi 指向下一个字符
213     mov cl, [put_int_buffer+edi]     ; 获取下一个字符到 cl 寄存器
214     cmp edi,8
215     jl .put_each_num
216     popad
217     ret
```

一眼望去代码 6-8 感觉还好，工作量也不算大，put_int 函数体只有 60 行左右。该函数的功能是将 32 位整型数字转换成字符后输出。函数转换实现的原理是按十六进制来处理 32 位数字，每 4 位二进制表示 1 位十六进制，将各十六进制数字转换成对应的字符，一共 8 个十六进制数字要处理。

在程序的第 5～6 行，用 section 定义了一个数据区，里面用伪指令 dq 申请了 8 字节的内存 put_int_buffer，它作为转换过程中用的缓冲区，实际上它的用途就是用于存储转换后的字符。

和大家交待下缓冲区为什么是 8 字节大小。咱们只支持 32 位数字的输出，按每 4 位二进制数为 1 位十六进制数计算，共 8 个十六进制数字要处理，每个数字虽然只是 4 位，但它们转换成对应的字符后，这些数字就得变成对应的 ASCII 码，ASCII 码是 1 字节大小，所以每个字符需要 1 字节的空间，这就是需要 8 字节缓冲区的原因。说一下伪指令 dq（它是由编译器提供的，并不是 CPU 支持的指令，所以称之伪指令），它用来定义操作数占用的字节数，q 是 quad 的简写，意为 4，定义 4 个字，也就是 8 个字节。

put_int 在第 160 行开始，第 161～164 行先是备份寄存器环境，这里还是用 pushad 指令备份全部 32 位通用寄存器。这里多说两句，与之前不同的是这里借鉴 C 调用约定，先把栈顶 esp 赋值给 ebp，再通过 ebp 来获取参数。这一方面是代码风格问题，另一方面这是 32 位保护模式下的改进，即内存寻址中支持用 esp 作为基址。之前咱们通过 esp 直接从栈中获取，通过那样的例子您知道函数中“push ebp”，“mov ebp，esp”，“mov xx，[ebp+n]”用这三步获取参数并不是必须的。不过话又说回来，直接通过 esp 获取参数是不太好的习惯，难免有压栈操作会改变 esp，用 ebp 就不同了，不显式改变它永远不会变。所以，尽管 32 位支持寄 esp 寻址，但还是建议通过 ebp 来获取参数，以后咱们也会这样做，当然这是细节问题，无需过多讨论。

在将参数获取到寄存器 eax 后，又将其送到 edx 中，这两个寄存器在后面要配合在一起使用。eax 寄存器是作为参数的备份，而 edx 寄存器是每次参与数位转换的寄存器，主要是由它做转换源，每当转换完 1 个十六进制数后，再由 eax 为其更新下一个待转换的数字。

第 165 行为 edi 赋值为 7，它表示在缓冲区中的偏移量，这里偏移为 7，表示指向缓冲区中最后一个 1 字节，目的是在该地址处存储数字最低 4 位二进制（也就是十六进制中的最低 1 位）对应的字符。

第 166 行为 ecx 赋值为 8，它表示要处理的数字的个数，32 位数字中，每 4 位二进制表示 1 个十六进制，十六进制数字的个数是 8，咱们按照十六进制来处理，所以共 8 个。

第 167 行把 ebx 作为缓冲区的基址，该地址就是前面已经定义的 put_int_buffer，我们把数字转换后的字符都存储在这里面。

第 169～188 行便是 put_int 的核心，将 32 位数字按照十六进制的形式从低位到高位逐个处理。

第 171 行，通过 and 与运算只保留数字的最低十六进制位，这是我们最先处理的部分，也就是先从 32 位数字的最低 4 位开始处理。接下来要将它转换成对应的字符。

下面说说转换的原理。

在 ASCII 表中，相同类别的字符是连续编码的，数字字符之间也都是连续的，比如字符'0'～'9'的 ASCII 码范围是 48～57，大写英语字母'A'～'Z'的 ASCII 码范围是 65～90。

将数字转换成字符的原理是用待转换的数字减去各自的起始数字得到偏移量，再用此偏移量加上对应字符所在类别的起始字符 ASCII 码。我知道这样描述很抽象，很费解，但我隐约觉得您至少明白了一点点，是不是我高估自己的表达能力了？哈哈，其实就是相对于字符 ASCII 码再加个增量，不过还是给大家举两个例子心里才踏实。

假设把数字 3 转换成字符，其过程是：用 3 减它的起始数字 0 得到偏移量 3，数字 3 对应的是字符'3'，字符'3'所在类别的起始字符是'0'，其 ASCII 码为 48，再用 3 加上 48，得到的 51 便是字符'3'的 ASCII 码。

假设把数字 E 转换成字符，其过程是：用 E 减去它的起始数字 A（10）得到偏移量 4，数字 E 对应的是字符'E'，字符'E'所在类别的起始字符是'A'，其 ASCII 码为 65，再用 4 加上 65，得到的 69 便是字符'E'的 ASCII 码。

虽然我前面表达得不清楚，但通过这两个例子，我想大家一定是没法再清楚了。

精简一下，将数字转换成字符的原理是：

- 如果是 0～9 之间的数字，用该数字加上字符'0'的 ASCII 码 48。
- 如果是 A～F 之间的数字，用该数字减去 10 后再加上字符'A'的 ASCII 码 65。

第 172 行开始判断该数字是否大于 9，大于 9 则属于 A～F 范围，否则属于 0～9。接下来到第 178 行的代码便是按照以上规则分别转换字符，这里为了语义清楚，对于字符'0'和字符'A'的 ASCII 码，在第 174 行和第 178 行直接写字符'0'和字符'A'，由编译器将其转换成各自的 ASCII 码。

这时候咱们的缓冲区 put_int_buffer 派上用场了，字符转换完成后就存储到这里。在由数字变成字符后，它在内存中的顺序可不能按照各位数字本身在内存中的顺序了（x86 架构是小端字符序，数字中的高位在内存的高地址，数字中的低位在内存的低地址），因为它们已经不再是数字了，处理它们的时候，都是以各字符（1 字节）的单位来处理的，一视同仁，不分高低。所以，咱们最好按照正常顺序来存储，高位的字符放在前面低地址，低位的字符放在后面高地址，这才是人习惯的自然顺序。不过这样一来，其存储顺序就有些类似大端字节序了，只不过咱们这里的是“大端字符序”，这个是我自己乱起的词，意思您懂了就行。

在第 182 行开始存储字符，edi 之前已经被赋值为 7，这是从后往前写字符，就是咱们的“大端字符序”。在第 184～185 行，通过“mov [ebx+edi]， dl”往 put_int_buffer 中写入转换好的字符，dec 指令使寄存器 edi 的值逐渐减少，使偏移量由高到低。

第 186 行通过 shr 右移指令把 eax 寄存器向右移动 4 位，去掉已转换完成的低 4 位，随后在第 187 行赋值给 edx，再进行下一轮的处理。这就是前面所说的 eax 和 edx 的“配合”。

代码第 192～202 行是准备跳过原来数字中高位的 0。如果待打印的数字其高位为 0，比如 0x00012345 这样的形式，我们在打印的时候应该将前面连续的多个 0 去掉，仅输出为 0x12345 才更人性化，注意，打

印到屏幕上的字符不包括十六进制的 0x 前缀。

第 193 行，edi 作为缓冲区中的偏移量，经过前面的转换之后，edi 此时已经为-1 了，即 0xffffffff，故通过 inc 指令使其恢复为 0，这也是为指向缓冲区中最低地址做准备。

第 199～202 行便是从最高位字符逐个与字符'0'比对，直到找出不为 0 的字符。寄存器 cl 用来存储字符 ASCII 码，表示当前处理的字符，edi 作为字符指针，用来指向缓冲区中的字符地址。

其中，第 199 行，由于缓冲区中的字符已经是按照“大端字符序”存储了，所以此时的 edi 作为偏移量，其值为 0，缓冲区中的偏移从 0 起便指向最高位字符。到这大家体会到第 193 行的 inc edi 指令使 edi 恢复为 0 的良苦用心了。

说一下第 203 行，这里将 edi 用 dec 指令减 1 的原因是通过上面第 199～202 行的代码，发现当前字符为非'0'时，edi 在第 200 行通过 inc 指令已经指向了下一个字符，为了使后面的打印方便，当前字符 cl 和字符指针 edi 应该是匹配的，所以在此将 edi 恢复成指向当前字符。

如果在最高位找到字符'0'，程序流程会到第 194 行的.skip_prefix_0，在这里会判断数字字符是否为全 0，这里的逻辑是偏移 edi 变成 8 时，表示已经找到了 8 个 0，所以判断为全 0。比如数字为 0x00000000 这种情况，其转换的字符会是'0"0"0"0"0"0"0"0'。随后会跳到.full0 处，也就是第 206～207 行，在那里将其处理为字符'0'。

程序执行到此，此时的 edi 指向缓冲区中左起第 1 个非'0'的字符，接下来的第 208～215 行会逐个打印后面的字符，这样就实现了打印字符串的目的。

第 216 行用 popad 指令恢复 32 位寄存器环境后在第 217 行结束返回。

好了，代码说完了，似乎大家也知道下一步该上机测试了，依旧是改头文件 print.h 和 main.c。先在 print.h 中增加 put_int 的声明，见代码 6-9。

代码 6-9　（project/c6/c/lib/kernel/print.h）

```
1 #ifndef __LIB_KERNEL_PRINT_H
2 #define __LIB_KERNEL_PRINT_H
3 #include "stdint.h"
4 void put_char(uint8_t char_asci);
5 void put_str(char* message);
6 void put_int(uint32_t num);   // 以十六进制打印
7 #endif
8
```

见代码 6-9 第 6 行，put_int 只支持 32 位整数输出，所以其参数是 uint32_t，它定义在 stdint.h 中，就是 unsigned int。

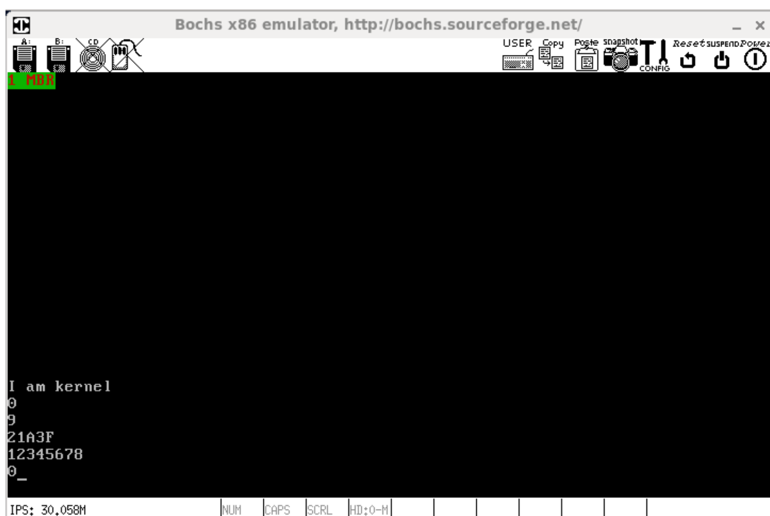
下面看 main.c，见代码 6-10。

代码 6-10　（project/c6/c/kernel/main.c）

```
 1 #include "print.h"
 2 void main(void) {
 3    put_str("I am kernel\n");
 4    put_int(0);
 5    put_char('\n');
 6    put_int(9);
 7    put_char('\n');
 8    put_int(0x00021a3f);
 9    put_char('\n');
10    put_int(0x12345678);
11    put_char('\n');
12    put_int(0x00000000);
13    while(1);
14 }
```

编译链接运行过程略，执行结果如图 6-11 所示。

图 6-11 中的运行结果，我目测是吻合的，这是咱们的最后一个打印函数，有关打印的部分到此就告一段落了，以后我们的 printf 函数也要基于这些函数来实现。好啦，兄弟们下节再见。



▲图 6-11　put_int 运行效果

6.4 内联汇编

之前和大家介绍过了一种汇编方法，就是 C 代码和汇编代码分别编译，最后通过链接的方式结合在一起形成可执行文件。

另一种方式就是在 C 代码中直接嵌入汇编语言，强大的 GCC 无所不能，咱们本节要学习的就是这一种，它称为内联汇编。

其实还有另外一种，就是那些技术大牛才能玩得转的方式，将 C 代码编译为汇编代码后，再修改汇编代码。

6.4.1　什么是内联汇编

内联汇编称为 inline assembly，GCC 支持在 C 代码中直接嵌入汇编代码，所以称为 GCC inline assembly。大家知道，C 语言不支持寄存器操作，汇编语言可以，所以自然就想到了在 C 语言中嵌入内联汇编提升“战斗力”的方式，通过内联汇编，C 程序员可以实现 C 语言无法表达的功能，这样使开发能力大为提升。

内联汇编按格式分为两大类，一类是最简单的基本内联汇编，另一类是复杂一些的扩展内联汇编，在介绍它们之前，其实还有一点点头疼的事，内联汇编中所用的汇编语言，其语法是 AT&T，并不是咱们熟悉的 Intel 汇编语法，GCC 只支持它，所以咱们还得了解下 AT&T。

6.4.2　汇编语言 AT&T 语法简介

我们在大学所学习的汇编语言大多数都是 Intel 语法，也许这和教学系统都是微软的操作系统 DOS 和 Windows 有关，翻翻过去的教材，一律全是 DOS 下汇编或 Windows 下汇编。Linux 内核中的汇编代码一般都是 AT&T 语法，我想，随着 Linux 普及，以后在教学中会越来越多采取 AT&T 语法啦。

什么是 AT&T 语法？

AT&T 是汇编语言的一种语法风格、格式。在某一处理器平台上，无论汇编代码是什么语法，其编译出来的机器码是一样的，所以不要误以为 AT&T 是一种新的机器语言。它仅仅是表达方式不同，意思是一样的，这就像咱们汉语中，比如，“我今天与贺亚涛在食堂吃饭”，“今天在食堂，贺亚涛和我一起吃饭”，表达的都是同一个意思。

AT&T 首先在 UNIX 中使用，可当初 UNIX 并不是在 x86 处理器上开发的，最初是在 PDP-11 机器上开发的，后来又移植到 VAX 和 68000 的处理器上，所以 AT&T 的语法自然更接近于这些处理器的特性。虽然 UNIX 后来又移植到 x86 上了，但还是要尊重 UNIX 圈内的习惯，其汇编语法接近于那些前辈处理器上的语法，这就是 AT&T 语法。

无论语法再怎么变，汇编语言中指令关键字肯定不能有太大出入，名字非常接近，只是在指令名字的最后加上了操作数大小后缀，b 表示 1 字节，w 表示 2 字节，l 表示 4 字节。比如压栈指令，Intel 中是 push，AT&T 中是 pushl，最后这个'l'表示压入 4 字节（long 型大小）。在了解 Intel 汇编指令的情况下，基本上能够看懂 AT&T 的汇编指令。它们的主要差别是语法风格，咱们对照着看下这两种风格的区别吧，见表 6-9。

表 6-9　Intel 和 AT&T 汇编风格对比

区别	Intel	AT&T	说　　明
寄存器	寄存器前无前缀	寄存器前有前缀%	
操作数顺序	目的操作数在左，源操作数在右	源操作数在左，目的操作数在右	Intel 的设计思想是目的操作数=源操作数，所以目的操作数为左值 AT&T 的设计思想是将源操作数->目的操作数，所以目的操作数在右边
操作数指定大小	有关内存的操作数前要加数据类型修饰符：byte 表示 8 位，word 表示 16 位，dword 表示 32 位，如 mov byte[0x1234]，eax	指令的最后一个字母表示操作数大小，b 表示 8 位，w 表示 16 位，l 表示 32 位。如 movl　%eax, var	在 AT&T 语法中，内存地址是第一位的，所以默认数字就是内存地址，操作数是数字就等同于操作数是内存，所以左边的 var 并没有像 Intel 语法那样加上中括号[]。如果是立即数则要加$前缀，这才表示普通的数字
立即数	无前缀，如 6	有前缀$，如$6	
远跳转	jmp far segment:offset	ljmp $segment:$offset	
远调用	call far segment:offset	lcall $segment:$offset	
远返回	ret far n	Iret $n	

以表 6-9 中未列出这两种语法在内存寻址方面的差异，个人觉得区别还是很大的，下面单独说说。

在 Intel 语法中，立即数就是普通的数字，如果让立即数成为内存地址，需要将它用中括号括起来，“[立即数]”这样才表示以“立即数”为地址的内存。

而 AT&T 认为，内存地址既然是数字，那数字也应该被当作内存地址，所以，数字被优先认为是内存地址，也就是说，操作数若为数字，则统统按以该数字为地址的内存来访问。这样，立即数的地位比较次要了，如果想表示成单纯的立即数，需要额外在前面加个前缀$。

无论是哪种汇编语言风格，都要有访问内存的能力，这就是内存寻址。

咱们之前学习了 Intel 汇编语法中的很多寻址方式，就内存寻址来说，有直接寻址、基址寻址、变址寻址、基址变址寻址。也可能是习惯了的原因，我个人觉得 Intel 语法真的很直白，容易理解，尤其是在和 AT&T 的内存寻址相比较之后……

而在 AT&T 中的内存寻址还是挺独特的，它的内存寻址有固定的格式。

segreg（段基址）：base_address(offset_address,index,size)

该格式对应的表达式为：

segreg（段基址）：base_address+ offset_address+ index*size。

此表达式的格式和 Intel 32 位内存寻址中的基址变址寻址类似，Intel 的格式：

segreg:[base+index*size+offset]

不过与 Intel 不同的是 AT&T 地址表达式的值是内存地址，直接被当作内存来读写，而不是普通数字。

看上去格式有些怪异，但其实这是一种“通用”格式，格式中短短的几个成员囊括了它所有内存寻址的方式，任意一种内存寻址方式，其格式都是这个通用格式的子集，都是格式中各种成员的组合。下面介绍下这些成员项。

base_address 是基地址，可以为整数、变量名，可正可负。

offset_address 是偏移地址，index 是索引值，这两个必须是那 8 个通用寄存器之一。

size 是个长度，只能是 1、2、4、8（Intel 语法中也是只能乘以这 4 个数）。

下面看看内存寻址中有哪些方式，注意，这些方式都是上面通用格式的一部分。

直接寻址：此寻址中只有 base_address 项，即后面括号中的内容全不要，base_address 便为内存啦，比如 movl $255，0xc00008F0，或者用变量名：mov $6，var。

寄存器间接寻址：此寻址中只有 offset_address 项，即格式为（offset_address），要记得，offset_address 只能是通用寄存器。寄存器中是地址，不要忘记格式中的圆括号，如 mov (%eax), %ebx。

寄存器相对寻址：此寻址中有 offset_address 项和 base_address 项，即格式为 base_address（offset_address）。这样得出的内存地址是基址+偏移地址之和。

各部分还是要按照格式填写，如 movb -4(%ebx),%al，功能是将地址（ebx-4）所指向的内存复制 1 字节到寄存器 al。

变址寻址：此类寻址称为变址的原因是含有通用格式中的变量 Index。因为 index 是 size 的倍数，所以有 index 的地方就有 size。既然是变址，只要有 index 和 size 就成了，base_address 和 offset_address 可有可无，注意，格式中没有的部分也要保留逗号来占位。一共有 4 种变址寻址组合，下面各举个例子。

无 base_address，无 offset_address：

movl %eax,(,%esi,2)

功能是将 eax 的值写入 esi*2 所指向的内存。

无 base_address，有 offset_address：

movl %eax,(%ebx,%esi,2)

功能是将 eax 的值写入 ebx+esi*2 所指向的内存。

有 base_address，无 offset_address：

movl %eax,base_value(,%esi,2)

功能是将 eax 的值写入 base_value+esi*2 所指向的内存。

有 base_address，有 offset_address：

movl %eax,base_value(%ebx,%esi,2)

功能是将 eax 的值写入 base_value+ebx+esi*2 所指向的内存。

好啦，AT&T 就简单介绍到这，咱们重点是内联汇编。

6.4.3 基本内联汇编

基本内联汇编是最简单的内联形式，其格式为：

```
asm [volatile] ("assembly code")
```

各关键字之间可以用空格或制表符分隔，也可以紧凑挨在一起不分隔，各部分意义如下：

关键字 asm 用于声明内联汇编表达式，这是内联汇编固定的部分，不可少。

asm 和__asm__是一样的，是由 gcc 定义的宏：#define __asm__ asm。

因为 gcc 有个优化选项-O，可以指定优化级别。当用-O 来编译时，gcc 按照自己的意图优化代码，说不定就会把自己所写的代码修改了。关键字 volatile 是可选项，它告诉 gcc：“不要修改我写的汇编代码，请原样保留”。volatile 和__volatile__是一样的，是由 gcc 定义的宏：#define __volatile__ volatile。

“assembly code”是咱们所写的汇编代码，它必须位于圆括号中，而且必须用双引号引起来。这是格式要求，只要满足了这个格式 asm [volatile] (“”)，assembly code 甚至可以为空。

下面说下 assembly code 的规则。

（1）指令必须用双引号引起来，无论双引号中是一条指令或多条指令。

（2）一对双引号不能跨行，如果跨行需要在结尾用反斜杠'\'转义。

（3）指令之间用分号'；'、换行符'\n'或换行符加制表符'\n''\t'分隔。

提醒一下，即使是指令分布在多个双引号中，gcc 最终也要把它们合并到一起来处理，合并之后，指令间必须要有分隔符。所以，当指令在多个双引号中时，除最后一个双引号外，其余双引号中的代码最后一定要有分隔符，这和其他编程语言中表示代码结束的分隔符是一样的，如：

```
asm(“movl $9,%eax;””pushl %eax”)          正确
asm(“movl $9,%eax””pushl %eax”)           错误
```

大家注意，在内联汇编中，咱们要注意操作数的顺序啦，现在是和 Intel 反着的。

给大家举个例子，见文件 inlineASM.c。

```
 1 char* str="hello,world\n";
 2 int count = 0;
 3 void main(){
 4 asm("pusha;  \
 5     movl $4,%eax; \
 6     movl $1,%ebx; \
 7     movl str,%ecx;\
 8     movl $12,%edx;\
 9     int $0x80;    \
10     mov %eax,count;\
11     popa          \
12     ");
13 }
```

代码 inlineASM.c 演示用汇编代码直接调用“系统调用”write 来打印字符串，该系统调用执行后会返回打印的字符数。

第 1～2 行定义了两个全局变量，待打印的字符串是 str，count 用来存储返回值。

第 4～12 行是内联汇编，这是咱们之前说过的 C 语言中跨过运行库直接调用系统调用的实例。这完全是 AT&T 风格的汇编语句：寄存器前面加前缀%，立即数前面加前缀$，操作数由左到右的顺序。似乎看上去很简单。

第 4 行将 8 个通用寄存器压栈，AT&T 中的汇编指令是 pusha（Intel 中的是 pushad）。

第 5 行传入第 4 号系统调用，这就是 write 的调用号。

第 6～8 行是为 write 系统调用传入参数，前面说系统调用的时候有讲过参数传递所用到的寄存器，不再赘述。

第 9 行用 int 0x80 执行系统调用，在 AT&T 中立即数的地位比较低，要加$前缀才表示数字为立即数（常数）。

第 10 行是获取 write 的返回值，返回值都是存储在 eax 寄存器中，所以将其复制到变量 count 中。

好啦，编译运行看结果，如图 6-12 所示。

```
[work@localhost book]$ gcc -o inlineASM.bin inlineASM.c
[work@localhost book]$ ./inlineASM.bin
hello,world
```

▲图 6-12　内联汇编运行结果

大家注意到没有，inlineASM.c 中的变量 count 和 str 定义为全局变量。对的，在基本内联汇编中，若要引用 C 变量，只能将它定义为全局变量。如果定义为局部变量，链接时会找不到这两个符号，这就是基本内联汇编的局限性，简单的东西往往功能不够强大，所以咱们还得学下扩展内联汇编形式，下一节走起。

6.4.4　扩展内联汇编

由于基本内联汇编功能太薄弱了，所以才对它进行了扩展以使其功能强大。不过，易用性往往与功能强弱是成正比的，如您所料，扩展内联汇编确实有点难，但在求知欲的驱使下，就让咱们痛并快乐着吧。

gcc 本身是个 C 编译器，要让其支持汇编语言，必然牵扯到以下问题。

- 在内联汇编代码插入点之前的 C 代码，其编译后也要被分配寄存器等资源，插入的汇编代码也要使用寄存器，这是否会造成资源冲突？
- 汇编语言如何访问 C 代码中的变量？

您看，内联汇编真不是简单地写两句汇编代码就完事了，所以，很多人宁可单独写纯汇编文件再链接，也不愿意写内联汇编。

我们从头分析，假设目前没有扩展内联汇编。

当汇编代码嵌入到 C 代码中，如果汇编代码想把 C 代码中的变量作为操作数加载到寄存器，如何找到可用的寄存器，这可是个大问题，程序员并不知道哪个寄存器已经被分配了，哪些寄存器是空闲的。即使知道了寄存器的分配情况也还不够，有些底层操作，对寄存器的要求是固定的（比如 in/out 指令，就得使用 al 作为数据寄存器），万一那个固定的寄存器已经被占用了，咱们在使用前还得把它备份。

也许您觉得，这些问题不难啊，我不管之前用了哪些寄存器，我在内联汇编中用哪些寄存器时就先将其入栈备份，用完了再恢复呗。听上去不错，但由用户来保证数据完整性简直就是灾难，人的精力是有限

的，谁知道会不会漏掉哪个寄存器呢，而且在出了问题后也不容易查出来。再说，运行中有大量的压栈操作，访问内存本身就比较慢，不如在编译阶段由编译器优化，直接分配给寄存器或用寄存器缓存，这样程序运行才更快。所以，这类事情还是交给编译器自己做这事才放心。

既然编译器对咱们不放心，那么现在的问题变成了如何将 C 代码中的变量变成汇编代码中的操作数，由于编译器无法预测用户的需求，这些只得让用户控制，故编译器采取的做法是它提供一个模板，让用户在模板中提出要求，其余工作由它负责实现。这些用户提出的要求，就是后面所说的约束。

因此，内联汇编的格式也变得让人生畏了，感觉既不像 C 语言，也不像汇编语言，似乎是一种中间产物，不信您看。

```
asm [volatile] (“assembly code”:output : input : clobber/modify)
```

和前面的基本内联汇编相比，扩展内联汇编在圆括号中变成了 4 部分，多了 output、input 和 clobber/modify 三项。其中的每一部分都可以省略，甚至包括 assembly code。省略的部分要保留冒号分隔符来占位，如果省略的是后面的一个或多个连续的部分，分隔符也不用保留，比如省略了 clobber/modify，不需要保留 input 后面的冒号。

assembly code：还是用户写入的汇编指令，和基本内联汇编一样。

汇编代码的运行是需要输入参数的，其运行之后也可产出结果。

在 C 代码中内嵌汇编的目的是让汇编帮助 C 完成某些功能，所以 C 代码就要为其提供参数和用于存放其输出结果的空间。这样一来，内联汇编代码类似机器，C 代码类似人。机器要运行，人就要为机器提供加工的源材料（input），机器运行后，将生产出来的成果放到人能够得着的地方（output），人才能获取机器的输出结果。input 和 output 正是 C 为汇编提供输入参数和存储其输出的部分，这是汇编与 c 交互的关键，我们之前的讨论就通过这两项解决。

output:output 用来指定汇编代码的数据如何输出给 C 代码使用。内嵌的汇编指令运行结束后，如果想将运行结果存储到 c 变量中，就用此项指定输出的位置。output 中每个操作数的格式为：

“操作数修饰符约束名”（C 变量名）

其中的引号和圆括号不能少，操作数修饰符通常为等号'='。多个操作数之间用逗号'，'分隔。

input：input 用来指定 C 中数据如何输入给汇编使用。要想让汇编使用 C 中的变量作为参数，就要在此指定。input 中每个操作数的格式为：

“[操作数修饰符] 约束名”（C 变量名）

其中的引号和圆括号不能少，操作数修饰符为可选项。多个操作数之间用逗号'，'分隔。

单独强调一下，以上的 output()和 input()括号中的是 C 代码中的变量，output（c 变量）和 input（c 变量）就像 C 语言中的函数，将 C 变量（值或变量地址）转换成汇编代码的操作数。

clobber/modify：汇编代码执行后会破坏一些内存或寄存器资源，通过此项通知编译器，可能造成寄存器或内存数据的破坏，这样 gcc 就知道哪些寄存器或内存需要提前保护起来，后面会展开细说。

assembly code 中引用的所有操作数其实是经过 gcc 转换后的复本，“原件”都是在 output 和 input 括号中的 c 变量，后面通过各种例子您就明白了。

上面所说的“要求”，在扩展内联汇编中称为“约束”，它所起的作用就是把 C 代码中的操作数（变量、立即数）映射为汇编中所使用的操作数，实际就是描述 C 中的操作数如何变成汇编操作数。这些约束的作用域是 input 和 output 部分，咱们看看这些约束是怎么体现的，约束分为四大类。

- 寄存器约束

寄存器约束就是要求 gcc 使用哪个寄存器，将 input 或 output 中变量约束在某个寄存器中。常见的寄存器约束有：

a：表示寄存器 eax/ax/al

b：表示寄存器 ebx/bx/bl

c：表示寄存器 ecx/cx/cl

d：表示寄存器 edx/dx/dl

D：表示寄存器 edi/di

S：表示寄存器 esi/si

q：表示任意这 4 个通用寄存器之一：eax/ebx/ecx/edx

r：表示任意这 6 个通用寄存器之一：eax/ebx/ecx/edx/esi/edi

g：表示可以存放到任意地点（寄存器和内存）。相当于除了同 q 一样外，还可以让 gcc 安排在内存中

A：把 eax 和 edx 组合成 64 位整数

f：表示浮点寄存器

t：表示第 1 个浮点寄存器

u：表示第 2 个浮点寄存器

下面咱们先暂停一下，体验一下基本内联汇编和扩展内联汇编的区别，用加法指令 addl 在两种方式下做个简单的加法运算。

先看下基本内联汇编，见文件 base_asm.c。

```
 1 #include<stdio.h>
 2 int in_a = 1, in_b = 2, out_sum;
 3 void main() {
 4     asm(" pusha;                  \
 5            movl in_a, %eax;       \
 6            movl in_b, %ebx;       \
 7            addl %ebx, %eax;       \
 8            movl %eax, out_sum;    \
 9           popa");
10     printf("sum is %d\n",out_sum);
11 }
```

加法指令的两个输入操作数是 in_a 和 in_b，输出和存储在变量 out_sum 中。以上代码我相信大家能独立看懂，大家注意，在第 5～6 行输入操作数 in_a 和 in_b 是分别手动 movl 到寄存器 eax 和 ebx 的。加法的和是在第 8 行 movl 到变量 out_sum 中的。

在基本内联汇编中的寄存器用单个%做前缀，在扩展内联汇编中，单个%有了新的用途，用来表示占位符（一会儿细讲），所以在扩展内联汇编中的寄存器前面用两个%做前缀。

再看下用扩展内联汇编是怎么做的，见文件 reg_constraint.c。

```
1 #include<stdio.h>
2 void main() {
3    int in_a = 1, in_b = 2, out_sum;
4    asm("addl %%ebx, %%eax":"=a"(out_sum):"a"(in_a),"b"(in_b));
5    printf("sum is %d\n",out_sum);
6 }
```

大伙儿注意，扩展内联汇编中寄存器前缀是两个%。

同样是为加法指令提供参数，in_a 和 in_b 是在 input 部分中输入的，用约束名 a 为 c 变量 in_a 指定了用寄存器 eax，用约束名 b 为 c 变量 in_b 指定了用寄存器 ebx。addl 指令的结果存放到了寄存器 eax 中，在 output 中用约束名 a 指定了把寄存器 eax 的值存储到 c 变量 out_sum 中。output 中的'='号是操作数类型修饰符，表示只写，其实就是 out_sum=eax 的意思。

不知大家有没有疑惑，output 和 input 中用的约束都是同一个，这是否会存在顺序混乱、数据覆盖？其实只要清楚操作数（约束对应的寄存器或内存）被赋值的顺序就明白了，肯定永远是输入（input）中的汇编操作数优先被赋值，汇编代码经过运行，最后才是为输出（output）中的汇编操作数赋值。拿第 4 行来说，output 和 input 中的约束都有 a，也就是都用寄存器 eax 来导入导出数值。肯定是 eax 先在 input 部分中被赋值为变量 in_a，此时 eax 作为输入参数第一次被赋值，在加法指令 addl 运行后直接把结果放到寄存器 eax 中，此时 eax 作为输出结果第二次被赋值，这样 eax 直接就是最终的输出啦。之后再处理 out_put，由于 eax 已经是汇编中用于输出的操作数了，编译器背后通过 mov 操作把 eax 的值传给 out_sum，这一点在后面很多例子中会看到。

另外，在 input 中不可能存在多个 c 变量的约束为同一个寄存器的情况，您懂的，一个寄存器无法同时容纳多个变量值，这样编译的时候也会报错。

通过对比，有没有觉得扩展内联汇编“炫”一些啦？它其实就是个模板，咱们把参数往里面套就行了，内部运作交给 gcc 处理。继续看其他约束啦。

- 内存约束

内存约束是要求 gcc 直接将位于 input 和 output 中的 C 变量的内存地址作为内联汇编代码的操作数，不需要寄存器做中转，直接进行内存读写，也就是汇编代码的操作数是 C 变量的指针。

m：表示操作数可以使用任意一种内存形式。

o：操作数为内存变量，但访问它是通过偏移量的形式访问，即包含 offset_address 的格式。

下面的文件 mem.c 用约束 m 为例。

文件 mem.c

```
1 #include<stdio.h>
2 void main() {
3    int in_a = 1, in_b = 2;
4    printf("in_b is %d\n", in_b);
5    asm("movb %b0, %1;"::"a"(in_a),"m"(in_b));
6    printf("in_b now is %d\n", in_b);
7 }
```

mem.c 的作用是变量 in_b 用 in_a 的值替换。in_b 最终变成 1。

第 5 行是内联汇编，把 in_a 施加寄存器约束 a，告诉 gcc 把变量 in_a 放到寄存器 eax 中，对 in_b 施加内存约束 m，告诉 gcc 把变量 in_b 的指针作为内联代码的操作数。

为演示内存约束，咱们在第 5 行用了个新的符号——“%1”，它是序号占位符，一会儿咱们会细说，在这里大家认为它代表 in_b 的内存地址（指针）就行了。

顺便说下，第 5 行对寄存器 eax 的引用：%b0，这是用的 32 位数据的低 8 位，在这里就是指 al 寄存器。如果不显式加字符'b'，编译器也会按照低 8 位来处理，但它会发出警告。

Warning: using '%al' instead of '%eax' due to 'b' suffix

程序运行结果如图 6-13 所示。

```
[work@localhost inline_ASM]$ ./mem.bin
in_b is 2
in_b now is 1
```

▲图 6-13　内存约束演示

咱们顺便再检验下 C 变量被处理为指针的情况，看看 mem.c 被转换成的汇编代码是什么。还是要用 gcc 的–S 参数帮忙：编译到汇编语言，不进行汇编和链接。

键入命令 gcc -S -o /tmp/mem.S ./mem.c 回车，生成的汇编代码输出在/tmp/mem.S，下面见代码 mem.S。

代码 mem.S

```
…略
10 main:
 11         pushl   %ebp               ;堆栈框架
 12         movl    %esp, %ebp
 13         andl    $-16, %esp
 14         subl    $32, %esp          ;预留局部变量空间
 15         movl    $1, 28(%esp)       ;变量 in_a 在栈中位置
 16         movl    $2, 24(%esp)       ;变量 in_b 在栈中位置
…略
 22         movl    28(%esp), %eax
;内联汇编的 input,将 in_a 赋值给 eax
 23 #APP
 24 # 5 "./mem.c" 1
 25         movb %al, 24(%esp);        ;内联汇编,直接将 al 写入 in_b 的内存
 26 # 0 "" 2
 27 #NO_APP
…略
```

这里只摘录了 mem.S 和内联汇编相关的部分。

第 11～14 行是有关堆栈框架的部分，我们知道局部变量是在栈中保存的，堆栈框架主要做的就是为局部变量在栈中分配空间。

第 15 行是将变量 in_a 存储到栈中距栈顶向上 28 字节的位置，并赋值为 1。

第 16 行是将变量 in_b 存储到栈中距栈顶向上 24 字节的位置，并赋值为 2。大伙先记住 in_b 等于 24（%esp），还记得这个格式吧，这是 AT&T 内存寻址方式之一：寄存器相对寻址。

第 23～27 行是咱们所写的内联汇编代码，#APP 和#NO_APP 之间的东东都是 gcc 加进去的。

第 25 行，直接将 al 寄存器 movb 到 in_b 的内存空间（栈中）。这验证了内存约束确实是把 C 变量的指针作为内联汇编代码的操作数。

另外提醒一下，内存约束也不是乱用的，至少在 assembly code 中的指令得允许操作数是内存，比如 asm("movl %0, %1;"::"m"(in_a),"m"(in_b))就会出问题。movl 指令不允许“内存”到“内存”的复制，编译阶段就会报错。

```
Error: too many memory references for `mov'
```

- 立即数约束

立即数即常数，此约束要求 gcc 在传值的时候不通过内存和寄存器，直接作为立即数传给汇编代码。由于立即数不是变量，只能作为右值，所以只能放在 input 中。

i：表示操作数为整数立即数

F：表示操作数为浮点数立即数

I：表示操作数为 0～31 之间的立即数

J：表示操作数为 0～63 之间的立即数

N：表示操作数为 0～255 之间的立即数

O：表示操作数为 0～32 之间的立即数

X：表示操作数为任何类型立即数

为节约篇幅，后面将立即数约束同其他约束一起演示，这里没有单独样例。

- 通用约束

0～9：此约束只用在 input 部分，但表示可与 output 和 input 中第 n 个操作数用相同的寄存器或内存。

为节约篇幅，后面会安排此约束的例子。

有关约束就说到这里，不过我怕没有将约束表达清楚，容小弟我再啰嗦十块钱的。

由于我们是在 C 语言中插入汇编代码，所以约束的作用是让 C 代码的操作数变成汇编代码能使用的操作数，所有的约束形式其实都是给汇编用的。故，约束是 C 语言中的操作数（变量或立即数）与汇编代码中的操作数之间的映射，它告诉 gcc，同一个操作数在两种环境下如何变换身份，如何对接沟通。编译过程中 C 代码是要先变成汇编代码的，内联汇编中的约束就相当于 gcc 让咱们指定 C 中数据的编译形式。在内联汇编中 assembly code 中用到的操作数，都是位于 output 和 input 中 C 操作数的副本，多数通过赋值的方式传给汇编代码，或者顶多是通过指针的形式，当操作数的副本在汇编中处理完成后，又重新赋值给 C 操作数。也可以这么说，C 操作数通过约束后，在汇编中的操作数是约束所指定的那个操作数载体，即内存或寄存器，如果是寄存器约束，汇编中操作的并不是 C 变量本身，而是 C 变量通过值传递到汇编的副本。举个例子，比如：

```
int in_a = 1, in_b = 2;
asm("movl %%eax, %%ebx"::"a"(in_a),"b"(in_b));
```

声明了两个 C 变量 in_a 和 in_b，在汇编代码中，表面上看是把变量 in_a 复制到了变量 in_b 中。但我们知道 movl 指令不能是从内存到内存的复制，所以，您一定知道或者早已经知道了，movl 的操作数是 C 变量 in_a 和 in_b 通过约束指定的操作数载体：寄存器 eax 和 ebx。

内存约束的讨论就到此为止，咱们该说点别的啦。

假设，我们用 a 指定寄存器为 eax，我们在汇编代码中可以用 eax 作为操作数。但有时我们对寄存器的要求并不严格，使用哪个都可以，所以我们可以用 r 来告诉 gcc 替我们自由安排。于是问题来了，由于 r 表示可以用 6 个寄存器之一，我们并不知道 gcc 为操作数分配了哪个寄存器。或者，我们对操作数用了内存约束，操作数没有名称可以引用，这时候我们在汇编代码中该如何引用操作数呢？

为方便对操作数的引用，扩展内联汇编提供了占位符，它的作用是代表约束指定的操作数（寄存器、内存、立即数），我们更多的是在内联汇编中使用占位符来引用操作数。

占位符分为序号占位符和名称占位符两种。

序号占位符

序号占位符是对在 output 和 input 中的操作数，按照它们从左到右出现的次序从 0 开始编号，一直到 9，也就是说最多支持 10 个序号占位符。

操作数用在 assembly code 中，引用它的格式是%0～9。

在操作数自身的序号前面加 1 个百分号'%'便是对相应操作数的引用。一定要切记，占位符指代约束所对应的操作数，也就是在汇编中的操作数，并不是圆括号中的 C 变量。

咱们还是拿前面文件 reg_constraint.c 的第 4 行举例，该行指令的功能是将 ebx 与 eax 相加后存储到 eax。代码如下：

```
asm("addl %%ebx, %%eax":"=a"(out_sum):"a"(in_a),"b"(in_b));
```

等价于

```
asm("addl %2, %1":"=a"(out_sum):"a"(in_a),"b"(in_b));
```

其中：

"=a"(out_sum)序号为 0，%0 对应的是 eax。

"a"(in_a)序号为 1，%1 对应的是 eax。

"b"(in_b)序号为 2，%2 对应的是 ebx。

由于扩展内联汇编中的占位符要有前缀%，为了区别占位符和寄存器，只好在寄存器前用两个%做前缀啦，这就是本节前面解释在扩展内联汇编中寄存器前面要有两个%做前缀的原因。

占位符所表示的操作数默认情况下为 32 位数据。指令的操作数大小并不一致，有的指令操作数大小是 32 位，有的是 16 位，有的是 8 位。当为这些指令提供操作数时，编译器会自动取 32 位数据的低 16 位给需要 16 位操作数的指令，取 32 位的低 8 位给需要 8 位操作数的指令。由于 32 位数据中，高 16 位没法直接使用，所以对于 16 位操作数只能取 32 位中的低 16 位。但对于 8 位操作数就不一样了，尽管默认情况下会用低 8 位（0～7 位）作为字节指令的操作数，但 32 位数据中能直接使用的字节不只是低 8 位，还有第 8～15 位，所以，对于字节指令，gcc 为我们提供了改变默认操作数的机会，我们可以自由选择是用 0～7 位，还是 8～15 位。这么说有点抽象，拿 32 位的寄存器 eax 举例，其常用的部分是 eax、ax、al（高 16 位没法直接用）。有些指令的操作数是字，所以用 ax 做操作数即可。有些指令操作数是字节，用 al 或 ah 都可以，默认情况下会将 al 当作操作数。这时候我们可以在%和序号之间插入字符'h'来表示操作数为 ah（第 8～15 位），或者插入字符'b'来表示操作数为 al（第 0～7 位）。

不知道大伙儿有没有稍感意外，怎么又冒出个字符'h'和字符'b'？其实它们并不孤独，它们的“同类”很多呢，这是属于机器模式中的内容，gcc 允许在更细的粒度上指定数据宽度或数据的某部分，有关机器模式的内容还是不少的，咱们在后面单独说吧。

下面先给大家演示序号占位符操作数情况，见文件 reg4.c。

文件 reg4.c

```
 1 #include<stdio.h>
 2 void main() {
 3    int in_a = 0x12345678, in_b = 0;
 4
 5    asm("movw %1, %0;":"=m"(in_b):"a"(in_a));
 6    printf("word in_b is 0x%x\n", in_b);
 7    in_b=0;    //将 in_b 恢复为 0,避免上次赋值造成混乱
 8
 9    asm("movb %1, %0;":"=m"(in_b):"a"(in_a));
10    printf("low byte in_b is 0x%x\n", in_b);
11    in_b=0;    //将 in_b 恢复为 0,避免上次赋值造成混乱
12
13    asm("movb %h1, %0;":"=m"(in_b):"a"(in_a));
14    printf("high byte in_b is 0x%x\n", in_b);
15 }
```

第 5 行是 movw 指令，往 in_b 中传入一个字，默认会传入低字，低 16 位：0x5678。

第 9 行是 movb 指令，往 in_b 中传入一个字节，默认会传入低字节，低 8 位：0x78。

第 13 行同样是 movb 指令，但这次我们在占位符中用了'h'，所以 in_b 中应该是 in_a 的第 7～15 位：0x56。运行结果如图 6-14 所示。

```
[work@localhost inline_ASM]$ ./reg4.bin
word in_b is 0x5678
low byte in_b is 0x78
high byte in_b is 0x56
```

▲图 6-14　序号占位符运行结果

由于序号占位符只支持 10 个操作数，虽然大多数情况下都够用了，但 gcc 还是提供了一种不受个数限制的占位符——名称占位符。

- 名称占位符

名称占位符与序号占位符不同，序号占位符靠本身出现在 output 和 input 中的位置就能被编译器辨识出来。而名称占位序需要在 output 和 input 中把操作数显式地起个名字，它用这样的格式来标识操作数：

[名称]”约束名”（C 变量）

这样，该约束对应的汇编操作数便有了名字，在 assembly code 中引用操作数时，采用%[名称]的形式就可以了。

这次我们用 8 位除法举例，见文件 reg5.c。

文件 reg5.c

```
1 #include<stdio.h>
2 void main() {
3    int in_a = 18, in_b = 3, out = 0;
4    asm("divb %[divisor];movb %%al,%[result]" \
5          :[result]"=m"(out)                  \
6          :"a"(in_a),[divisor]"m"(in_b)       \
7          );
8    printf("result is %d\n",out);
9 }
```

我们的目的是用 18 除以 3，最后打印结果是 6。

在第 6 行，被除数 in_a 通过寄存器约束 a 存入寄存器 eax，除数 in_b 通过内存约束 m 被 gcc 将自己的内存地址传给汇编做操作数，咱们不用关心此内存地址是哪里。为了引用除数所在的内存，我们用名称占位符标识它，名字是 divisor。这样汇编代码中，第 4 行除法指令 divb 可以通过%[divisor]引用除数所在的内存，进行除法运算。divb 是 8 位除法指令，商存放在寄存器 al 中，余数存放在寄存器 ah 中。所以第 4 行中用 movb 指令将寄存器 al 的值写入用于存储结果的 c 变量 out 的地址中。运行结果如图 6-15 所示。

```
[work@localhost inline_ASM]$ ./reg5.bin
result is 6
```

▲图 6-15　名称占位符

强调与总结：

无论是哪种占位符，它都是指代 C 变量经过约束后、由 gcc 分配的对应于汇编代码中的操作数，和 C 变量本身无关。这个操作数就是通过约束名所指定的寄存器、内存、立即数等，最终编译器要将占位符转换成这三种操作数类型之一。

在约束中还有操作数类型修饰符，用来修饰所约束的操作数：内存、寄存器，分别在 ouput 和 input 中有以下几种。

在 output 中有以下 3 种。

=：表示操作数是只写，相当于为 output 括号中的 C 变量赋值，如=a(c_var)，此修饰符相当于 c_var=eax。

+：表示操作数是可读写的，告诉 gcc 所约束的寄存器或内存先被读入，再被写入。

&：表示此 output 中的操作数要独占所约束（分配）的寄存器，只供 output 使用，任何 input 中所分配的寄存器不能与此相同。注意，当表达式中有多个修饰符时，&要与约束名挨着，不能分隔。

在 input 中：

%：该操作数可以和下一个输入操作数互换。

一般情况下，input 中的 C 变量是只读的，output 中的 C 变量是只写的。

修饰符'='只用在 output 中，表示 C 变量是只写的，功能相当于 output 中的 C 变量=约束的汇编操作数，如”=a”(c_var)，相当于 c_var=eax 的值。前面我们有了很多例子，不再单独演示。

修饰符'+'也只用在 output 中，但它具备读、写的属性，也就是它既可作为输入，同时也可以作为输出，所以省去了在 input 中声明约束。下面通过实例演示，见文件 reg6.c。

文件 reg6.ce

```
1 #include <stdio.h>
2 void main() {
3     int in_a = 1, in_b = 2;
4     asm("addl %%ebx, %%eax;":"+a"(in_a):"b"(in_b));
5     printf("in_a is %d\n", in_a);
6 }
```

运行结果如图 6-16 所示。

```
[work@localhost inline_ASM]$ ./reg6.bin
in_a is 3
[work@localhost inline_ASM]$
```

▲图 6-16　修饰符+演示

另一个'+'常用的场合是在“rep+字符串操作指令+cld 或 std”指令组合中，原因是在字符串操作指令中，esi 作为源变址，保存数据所在的源地址，在字符串操作指令执行前，esi 要作为数据源地址，所以当作参数被读入，成了输入对象，在字符串操作指令执行后，esi 也要被更新为下一个数据源的地址，此时 esi 又被写入，成了输出对象，所以 esi 是先被读入后被写入，非常适合用'+'来修饰。edi 作为目的变址，保存数据写入的目的地址。在字符串操作指令执行前，edi 要作为数据写入的目的地址，所以当作参数被读入，成了输入对象，在字符串操作指令执行后，edi 也要被更新为下一个数据所写入的目的地址，此时 edi 又被写入，成了输出对象，所以 edi 也是先被读入，后被写入，非常适合用'+'来修饰。

注意，常见的字符串操作指令有 movs[bwd]、ins[bwd]和 outs[bwd]、lods[bwd]和 stos[bwd]。但是，esi 和 edi 并不是被以上三组指令同时使用，只有 movs[bwd]才同时使用 esi 和 edi，它是把 esi 所指向的地址处的数据复制到 edi 所指向的内存地址处。ins[bwd]是从端口读入数据到内存的目的地址，故只涉及到 edi。outs[bwd]是把内存中的源数据写入端口，故只涉及到 esi。lods[bwd]是把内存中的源数据加载到寄存器 al、ax 或 eax，故只涉及到 esi。stos[bwd]是将寄存器 al、ax 或 eax 中的值写入内存中的目的地址，故只涉及到 edi。以上字符串指令每执行一次，所涉及到的源变址寄存器 esi 或目的变址寄存器 edi 都要根据操作数大小有所增减，至于是增加，还是减少，要取决于标志寄存器中的方向位 DF，若 DF 为 0，esi 和 edi 都自增，地址值越来越大，否则 DF 为 1，esi 和 edi 都自减，地址值越来越小。这些字符串操作指令在读写数据时，esi 和 edi 作为它们的输入操作数，执行完成后，根据 DF 位的情况自增或自减，这时又作为输出。以后在用内联汇编写 IO 端口函数时会用到，此处不再单独举例。

修饰符'&'用来表示此寄存器只能分配给 output 中的某个 C 变量使用，不能再分给 input 中某变量了。函数在执行完成时，返回值会存储在寄存器 eax 中。通常我们会将返回值获取到 C 变量中再处理。如果是让 gcc 自由分配寄存器，gcc 有可能把 eax 分配出去另做他用，有可能的一种情况是先调用函数，函数返回后又执行了其他操作，这个操作中用到了 gcc 分配的 eax 寄存器，于是函数的返回值便被破坏，见文件 reg7.c。

文件 reg7.c

```
1 #include <stdio.h>
 2 void main() {
 3    int ret_cnt = 0, test = 0;
 4    char* fmt = "hello,world\n";    // 共 12 个字符
 5    asm(" pushl %1;           \
 6      call printf;            \
 7      addl $4, %%esp;         \
 8      movl $6, %2"            \
 9      :"=a"(ret_cnt)          \
10      :"m"(fmt),"r"(test)     \
11      );
12     printf("the number of bytes written is %d\n", ret_cnt);
13 }
```

本文件的功能是打印字符串 hello，world 回车，然后再打印出 printf 的返回值，正常情况下 printf 会返回打印的字符数，”hello，world\n”共 12 个字符。

第 4 行定义了字符串”hello，world\n”的指针 fmt，这是给 printf 的参数，变量 test 是演示用的，用它来干扰返回值，没实际意义。

第 10 行通过内存约束 m 把字符串指针 fmt 传给了汇编代码，把变量 test 用寄存器约束 r 声明，由 gcc

自由分配寄存器。

第 5～6 行调用 printf。将第 4 行定义的字符串指针 fmt 通过压栈传参给 printf 函数，在第 6 行执行会屏幕会打印 hello，world 换行。在此，eax 寄存器中值是 printf 的返回值，应该为 12。

在第 7 行回收参数所占的栈空间。第 8 行把立即数 6 传入了 gcc 为变量 test 分配的寄存器。在第 9 行，output 部分中的 c 变量 ret_cnt 获得了寄存器的值。

程序运行结果如图 6-17 所示。

```
[work@localhost inline_ASM]$ gcc -o reg7.bin reg7.c
[work@localhost inline_ASM]$ ./reg7.bin
hello,world
the number of bytes written is 6
```

▲图 6-17　修饰符&演示 a

程序打印出了两行，第一行是正确的，但第二行我们原本想着打印返回值，应该是 12，而不是 6，并未按照我们的预期打印 the number of bytes written is 12。

这说明在文件 reg7.c 的第 8 行，%2 被 gcc 分配为寄存器 eax 了。

最好的证明就是看看 reg7.c 翻译为汇编后的样子，还是用到前面介绍的 gcc-S 命令，部分汇编代码见文件 reg7.S。

文件 reg7.S

```
…略
16        movl    $0, 24(%esp)        // 变量 ret_cnt
 17        movl    $0, 28(%esp)       // 变量 test
 18        movl    $.LC0, 20(%esp)
 19        movl    28(%esp), %eax
 20 #APP
 21 # 5 "reg7.c" 1
 22    pushl 20(%esp);  call printf; addl $4, %esp;movl $6, %eax
 23 # 0 "" 2
 24 #NO_APP
 25        movl    %eax, 24(%esp)
…略
```

大家直接看#APP 和#NO_APP 之间的部分，这部分是我们的内联汇编代码所在，第 22 行，在 call printf 后，把立即数 6 movl 到了寄存器 eax，这破坏了 printf 的返回值。

这时候修饰符'&'就派上用场了，只要为 test 约束的寄存器不要和 ret_cnt 相同就行，可以在 reg7.c 的第 9 行加个'&'，修改后的文件为 reg8.c。

文件 reg8.c

```
 1 #include <stdio.h>
 2 void main() {
 3    int ret_cnt = 0, test = 0;
 4    char* fmt = "hello,world\n";    // 共 12 个字符
 5    asm(" pushl %1;          \
 6      call printf;      \
 7      addl $4,%%esp;          \
 8      movl $6, %2"       \
 9      :"=&a"(ret_cnt)     \
10      :"m"(fmt),"r"(test)  \
11      );
12      printf("the number of bytes written is %d\n", ret_cnt);
13 }
```

其与 reg7 的区别就是第 9 行多加了个'&'。编译运行，结果如图 6-18 所示。

```
[work@localhost inline_ASM]$ gcc -o reg8.bin reg8.c
[work@localhost inline_ASM]$ ./reg8.bin
hello,world
the number of bytes written is 12
[work@localhost inline_ASM]$
```

▲图 6-18　修饰符&演示 b

下面用实例演示修饰符'%'的用法，顺便再把前面立即数约束和通用约束都揉到一起演示。

修饰符'%'表示 input 中的输入可以和下一个 input 操作数互换，通常用在计算结果与操作数顺序无关的指令中，比如加法和乘法，这里咱们用加法指令举例，见文件 reg9.c。

文件 reg9.c

```
1 #include<stdio.h>
 2 void main() {
```

```
3    int in_a = 1, sum = 0;
4    asm("addl %1, %0;":"=a"(sum):"%I"(2),"0"(in_a));
5    printf("sum is %d\n ", sum);
6 }
```

文件 reg9.c 演示了修饰符'%'、立即数约束、通用约束的用法，纯粹演示，并无实际意义。

第 4 行 input 部分中的"%I"（2），表示传了一个立即数 2，并用了立即数约束 I，这会让 gcc 将数字 2 变成汇编语言中的立即数$2。在这个输入中，还用到了修饰符'%'，这表示约束 I 对应的操作数可以和下一个输入所约束的操作数对换位置。下一个输入是"0"（in_a），前面用了通用约束'0'，这表示，要求 gcc 把分配给 C 变量 in_a 的操作数（寄存器或内存）同序号 0 对应的汇编操作数一样，也就是，位于 output 和 input 中序号为 0 的输入，其所约束的操作数（寄存器或内存）是什么，就把我安排成什么。序号 0 是"=a"（sum），约束 a 将寄存器 eax 分配给变量 sum，所以 in_a 也被分配为 eax，这有点类似修饰符'+'同时可读可写。

光看我这么说似乎还不够，咱们眼见为识，看看 reg9.c 生成的汇编代码是什么，见文件 reg9.S。

文件 reg9.S

```
…略
13        movl    $1, 24(%esp)        // in_a
 14        movl    $0, 28(%esp)        // sum
 15        movl    24(%esp), %eax      // in_a 写入寄存器 eax
 16 #APP
 17 # 4 "reg9.c" 1
 18        addl $2, %eax;              // $2 对应 input 中的"%I"(2)
 19 # 0 "" 2
 20 #NO_APP
 21        movl    %eax, 28(%esp)      //由于"=a"(sum)决定将 eax 存入 sum
…略
```

第 13 行栈中 24（%esp）的位置是变量 in_a。

第 14 行栈中 28（%esp）的位置是变量 sum。

第 15 行是把变量 in_a 写入寄存器 eax，这与文件 reg9.c 第 4 行中的"=a"(sum)和"0"(in_a)是吻合的，即 in_a 与 sum 所用的寄存器是一样的，都是 eax。

第 21 行将第 18 行的计算结果写入到 sum 变量中。

好啦，该说的都说啦，运行结果如图 6-19 所示。

```
[work@localhost inline_ASM]$ gcc -o reg9.bin reg9.c
[work@localhost inline_ASM]$ ./reg9.bin
sum is 3
 [work@localhost inline_ASM]$
```

▲图 6-19　修饰符%和约束

在最后，我们说下扩展内联汇编中的 clobber/modify 部分，这部分用于通知 gcc，我们修改了哪些寄存器或内存。

由于我们在 C 程序中嵌入了汇编代码，这必然会造成一些资源的破坏，本来嘛，人家 C 代码翻译后也要用到寄存器，突然来了一堆抢寄存器用的汇编指令，这肯定会使 gcc 重新为 C 代码安排寄存器等资源。为了解决资源冲突，我们得让 gcc 知道，我们改变了哪些寄存器或内存，这样 gcc 才能合理安排。

如果在 output 和 input 中通过寄存器约束指定了寄存器，gcc 必然会知道这些寄存器会被修改，所以，需要在 clobber/modify 中通知的寄存器肯定不是在 output 和 input 中出现过的。

也许您会认为，牵扯到修改寄存器或内存的部分，只差 assembly code 没说了，这部分还需要咱们明确告诉编译器吗？编译器不会自己扫描汇编指令？用到了哪些寄存器它还不知道吗？是的，gcc 还真不是很有把握知道哪些资源会被修改。在“明处”的指令确实可以检测到所修改的资源，比如 incl %%eax，eax 寄存器被+1 了，这种明显的改变 gcc 当然能扫描出来。可“暗处”的指令就无法保证了，比如在汇编中调用了一个函数，该函数内部会修改一些资源，或者该函数中又调用了其他函数，这保不准在哪一层调用有修改资源的代码，简直无法跟踪。所以必须要人为显式地告诉 gcc 我们动了哪些资源，这个资源就是寄存器和内存。

怎样通知 gcc 我们修改了哪些寄存器？

这个很简单，只要在 clobber/modify 部分明确写出来就行了，记得要用双引号把寄存器名称引起来，多个寄存器之间用逗号','分隔，这里的寄存器不用再加两个'%'啦，只写名称即可，如：

```
asm("movl %%eax, %0;movl %%eax,%%ebx":"=m" (ret_value)::"bx")
```

大家看，虽然修改的是寄存器 ebx，但只要在 clobber/modify 声明 bx 就可以了，甚至可以声明 al。原因是即使寄存器只变动一部分，它的整体也会全跟着受影响，所以在 clobber/modify 中声明寄存器时，可

以用低 8 位名称、低 16 位名称或全 32 位名称，如”al”/”ax”/”eax”都是指 eax 寄存器，其他通用寄存器也是一样的，不再举例。

如果我们的内联汇编代码修改了标志寄存器 eflags 中的标志位，同样需要在 clobber/modify 中用”cc”声明。

如果我们修改了内存，我们需要在 clobber/modify 中”memory”声明。

如果我们在 output 中使用了内存约束，gcc 自然会得到哪块内存被修改。但如果被修改的内容并未在 output 中，我们就需要用”memory”告诉 gcc 啦。

举个例子，还记得之前咱们所说的复制大块数据的三剑客指令吗？字符串指令 movsd 配合 cld 和 rep，通过指针 esi 和 edi 的不断变化使源数据被复制到新目的地，后来数据被复制到了哪里，修改了哪些内存，gcc 可就不知道了，咱得主动用”memory”坦白才行。

另外一个用”memory”声明的原因就是清除寄存器缓存。

内存相对寄存器来说还是比较慢的，gcc 为了提速，编译中有时会把内存中的数据缓存到寄存器，之后的处理都是直接读取寄存器。编译过程中编译器无法检测到内存的变化，只有编译出来的程序在实际运行中才会出现变量的值被改变，也就是出现了内存变化的情况。您想，如果程序在编译阶段就能检测程序行为的话，那还运行程序干吗，直接在该程序的编译阶段输出程序运行结果不就完了吗^_^。于是问题来了，编译器编译程序时，将变量的值缓存到寄存器。程序在运行时，当变量所在的内存有变化时，寄存器中的缓存还是旧数据，运行结果肯定就错了。于是，gcc 也为我们提供了选项，用来设置是否将变量缓存到寄存器中，这个选项就是 C 语言中的关键字 volatile，它表示该变量是不稳定的，容易被改变，这样 gcc 就不会将其缓存到寄存器。注意啦，这个关键字并不是内联汇编 asm 后面的可选项[volatile]，虽然名字是一样的，但汇编中的 volatile 是定义的宏 #define__volatile__volatile，在编译前的预处理阶段，汇编中的 volatile 最终会变成__volatile__，这和 C 语言中的 volatile 不冲突。用 C 语言中的 volatile 定义的变量，编译器就不会将该变量的值缓存到寄存器中，每次访问该变量时都会老老实实地从内存中获取。也就是说，只要编译器知道变量所在的内存有变化，它就会放弃寄存器缓存，到内存中取数据。

利用这个原理，不管变量的值是否会被编译器缓存到寄存器中，当我们需要绕过寄存器缓存，也就是希望读取到内存中最新的数据时，我们就可以在内联汇编中的 clobber/modify 部分用”memory”声明，通知编译器变量所在的内存数据变啦，这样它就会从内存再读取一次新数据啦。当然我们也可以在 C 代码中用 volatile 去修饰所定义的变量，但变量多了就有些麻烦了，所以还是用”memory”来声明更加方便。

6.4.5　扩展内联汇编之机器模式简介

下面介绍下 GCC 中的机器模式。

我知道，如果从未接触过机器模式这方面内容，无论我再怎样挥舞文字，都无法轻易地让大伙儿明白它的概念。所以，咱们循序渐进，慢慢引出它的来历，通过实例慢慢看。

在前面介绍序号占位符的时候，咱们已经引出了机器模式的内容：为了指定寄存器中的某部分，咱们引用了字符'h'和字符'b'，它们分别用来指定寄存器的第 8～15 位和低 8 位，这只是机器模式的用途之一。

比如寄存器约束 a 表示寄存器 al、ax、eax，可以在序号占位符中增加前缀字符 h 和 b 来引用寄存器 ah 和 al。不过这次我不想指定 ah 或 al 啦，如果我想指定 ax 或 eax，怎么做？为了回答这个问题，咱们再举个例子，见文件 mach_mode_warn.c。

文件 mach_mode_warn.c

```
1 #include<stdio.h>
2 void main() {
3    int in_a = 0x1234, in_b = 0;
4    asm("movw %1, %0":"=m"(in_b):"a"(in_a));
5    printf("in_b now is 0x%x\n", in_b);
6 }
```

这段代码很简单，目的是把 in_a 的低 16 位复制到 in_b 中。

第 4 行中，变量 in_a 的约束是 a，这表示由 gcc 把 in_a 的值分配给寄存器 al、ax 或 eax。这很模糊，

到底 gcc 把 in_a 的值分配给谁了呢？之后的 movw 指令也很模糊，我们只能这样理解：movw 指令将 al、ax 或 eax 中的 2 个字节复制到 in_b 所在的内存中（当然 al 中不可能有 2 个字节的数据）。有没有疑问，movw 是移动 2 个字节的数据，那 movw 的源操作数到底是不是 ax？编译一下就知道了。编译过程如图 6-20 所示。

```
[work@localhost inline_ASM]$ gcc -o mach_mode_warn.bin mach_mode_warn.c
mach_mode_warn.c: Assembler messages:
mach_mode_warn.c:4: Warning: using `%ax' instead of `%eax' due to `w' suffix
[work@localhost inline_ASM]$
```

图 6-20　机器模式举例 1

编译器发出了警告，大意是由于 w 前缀，用寄存器 ax 代替 eax。

大家应该注意到了图右下部分的框框，里面有个字符 w，实际代码中我们并没有添加 w，这说明默认情况下，gcc 用占位符引用操作数的时候，根据指令操作数大小的不同，添加了适当的前缀。验证一下这个想法，将上面第 4 行的代码中的%1，修改为%w1，结果为文件 mach_mode.c。

文件 mach_mode.c

```
 1 #include<stdio.h>
 2 void main() {
 3    int in_a = 0x1234, in_b = 0;
 4    asm("movw %w1, %0":"=m"(in_b):"a"(in_a));
 5    printf("in_b now is 0x%x\n", in_b);
6 }
```

这次的编译过程未报任何错误，运行结果如图 6-21 所示。

显然结果是正确的。这里的字符 w 是怎么回事呢？这是我们新接触的另一个控制字符，听兄弟我慢慢道来吧。

```
[work@localhost inline_ASM]$ ./mach_mode.bin
in_b now is 0x1234
[work@localhost inline_ASM]$
```

▲图 6-21　机器模式举例 2

w 和 h、b 一样，都是操作码，用来指代某种机器模式类型。

咱们看看源文件中的解释吧，看官方的说明更权威，以下内容来自文件 gcc-4.9.0/gcc/config/ i386/i386.c。

文件 i386.c

```
行号操作码描述
14741 /* Meaning of CODE:
14742     L,W,B,Q,S,T -- print the opcode suffix for specified size of operand.
14743     C -- print opcode suffix for set/cmov insn.
14744     c -- like C, but print reversed condition
14745     F,f -- likewise, but for floating-point.
14746     O -- if HAVE_AS_IX86_CMOV_SUN_SYNTAX, expand to "w.", "l." or "q.",
14747            otherwise nothing
14748     R -- print embeded rounding and sae.
14749     r -- print only sae.
14750     z -- print the opcode suffix for the size of the current operand.
14751     Z -- likewise, with special suffixes for x87 instructions.
14752     * -- print a star (in certain assembler syntax)
14753     A -- print an absolute memory reference.
14754     E -- print address with DImode register names if TARGET_64BIT.
14755     w -- print the operand as if it's a "word" (HImode) even if it isn't.
14756     s -- print a shift double count, followed by the assemblers argument
14757            delimiter.
14758     b -- print the QImode name of the register for the indicated operand.
14759            %b0 would print %al if operands[0] is reg 0.
14760     w --  likewise, print the HImode name of the register.
14761     k --  likewise, print the SImode name of the register.
14762     q --  likewise, print the DImode name of the register.
14763     x --  likewise, print the V4SFmode name of the register.
14764     t --  likewise, print the V8SFmode name of the register.
14765     g --  likewise, print the V16SFmode name of the register.
14766     h -- print the QImode name for a "high" register, either ah, bh, ch or dh.
14767     y -- print "st(0)" instead of "st" as a register.
14768     d -- print duplicated register operand for AVX instruction.
14769     D -- print condition for SSE cmp instruction.
14770     P -- if PIC, print an @PLT suffix.
14771     p -- print raw symbol name.
```

```
14772      X -- don't print any sort of PIC '@' suffix for a symbol.
14773      & -- print some in-use local-dynamic symbol name.
```

在文件 i386.c 的第 14755 行，这就是操作码 w 的意义，它表示：即使操作数不是一个字（2 字节）那样的大小，也要像一个字那样把它打印出来。简而言之就是不管操作数多少个字节，只打印 2 字节。

其中值得一提的是对于操作码 w 的解释中包含了"word" (HImode)，意思是说这个陌生的词“HImode”等同于 word，即 2 字节。在其他的操作码说明中，也有 SImode 或 QImode 等，这些都是什么呢？

以上以 mode 结尾的这些词实际上就是具体的机器模式名称。

什么是机器模式？咱们还是看官方文件是怎么解释的。在 GCC 源文件 gcc/machmode.def 中有这样一句说到：

A machine mode specifies a size and format of dataat the machine level.

机器模式用来在机器层面上指定数据的大小及格式。

用我自己的理解是 GCC 支持内联汇编，由于各种约束均不能确切地表达具体的操作数对象，所以引用了机器模式，用来从更细的粒度上描述数据对象的大小及其指定部分。

这些模式定义在哪里？我能看到吗？

在文件 gcc/machmode.def 中有这样一句话：This file defines only those modes which are of use on almost allmachines. Other modes can be defined in the target-specificmode definition file，　config/ ARCH/ARCH-modes.def。

GCC 根据不同的硬件平台，将机器模式定义在多个文件中，其中所有平台都通用的机器模式定义在 gcc/machmode.def 文件中，其他与具体平台相关的机器模式定义在自己的平台路径下，它们在 config/平台名称/平台名称-modes.def 文件中，如 config/i386/i386-modes.def。

机器模式是用枚举类型 enum machine_mode 来定义的，而实际上，以上所说的所有.def 文件中并没有现成的机器模式定义。原因是这样的，各平台下的.def 文件需要和 machmode.def 一起通过后端工具 genmodes 处理之后才能生成完整的机器模式数据，由于小弟我对这方面了解不多，也只能蜻蜓点水解释到此，有关此方面的内容，有兴趣的同学请自行研究。

不过，为满足大伙儿的好奇心，我还是找了个机器模式定义的样例给大伙儿，图 6-22 所示内容来自文件 gcc-4.9.0/gcc/testsuite/gcc.dg/vect/pr48765.c。

```
10 enum machine_mode
11 {
12   VOIDmode, QImode, HImode, PSImode, SImode, PDImode, DImode, TImode, OImode,
13    QFmode, HFmode, TQFmode, SFmode, DFmode, XFmode, TFmode, SCmode, DCmode,
14    XCmode, TCmode, CQImode, CHImode, CSImode, CDImode, CTImode, COImode,
15    BLKmode, CCmode, CCEVENmode, MAX_MACHINE_MODE
16 };
```

▲图 6-22　机器模式枚举定义

机器模式名称的结构大致是这样的：数据大小+数据类型+mode，比如 QImode，表示 QuarterInteger，即四分之一整型。QFmode 表示 QuarterFloating，即四分之一浮点型。

咱们只用到了和整数相关的机器模式，所以摘几个看看它们的意义，见表 6-10。

表 6-10　整型机器模式

机器模式名称	描　述
BImode	Bit，即比特模式，表示单个比特
QImode	QuarterInteger，即四分之一整型模式，表示 1 个字节的整数
HImode	HalfInteger，即半整型模式，表示 2 个字节的整数
SImode	Single Integer，即单整型模式，表示 4 个字节的整数
PSImode	Partial Single Integer，即部分单整型模式，表示 4 字节整型数，但只使用了部分字节
DImode	Double Integer，即双倍整型模式，表示 8 字节的整数
PDImode	Partial Double Integer，即部分双倍整型模式，表示 8 字节的整型数，但只使用了部分字节
TImode	Tetra Integer，即四倍整型模式，表示 16 字节的整数
OImode	Octa Integer，即八倍整型模式，表示 32 字节的整数

了解了机器模式后，咱们再返回去看文件 i386.c 中的操作码。实际上操作码就是指定操作数为寄存器中的哪个部分。咱们这里不用关注太多，初步了解 h、b、w、k 这几个操作码就够了，以后有需要时再说。

寄存器按是否可单独使用，可分成几个部分，拿 eax 举例。

- 低部分的一字节：al
- 高部分的一字节：ah
- 两字节部分：ax
- 四字节部分：eax

h –输出寄存器高位部分中的那一字节对应的寄存器名称，如 ah、bh、ch、dh。

b –输出寄存器中低部分 1 字节对应的名称，如 al、bl、cl、dl。

w –输出寄存器中大小为 2 个字节对应的部分，如 ax、bx、cx、dx。

k –输出寄存器的四字节部分，如 eax、ebx、ecx、edx。

对目前咱们的应用，以上几个操作码够用了，机器模式这块到这就结束了。

在机器模式介绍完后，个人觉得有关内联汇编这块已经说得不少了，足够应付今后的应用了，本节到此结束。

第 7 章　中断

就在我写本书的时候，我女朋友王小兔问我要不要跟她一起去超市，她知道这是我最喜欢的生活。每次和她一起逛超市都是一种享受，让我更加热爱生活，这使我觉得很幸福。当我们抱着最爱吃的各种饼干、巧克力，骑着单车从超市回来之后，我又开始打开电脑，继续写书了。

在我处理完逛超市这一“优先级更高”的事情后，还能回来继续写书，这说明这件事我的脑子里记着呢。这就是“中断”在生活中的实例之一。

这一节，我们将讲述中断。

7.1 中断是什么，为什么要有中断

中断是什么？这里所说的是发生在计算机世界里的中断。

由于 CPU 获知了计算机中发生的某些事，CPU 暂停正在执行的程序，转而去执行处理该事件的程序，当这段程序执行完毕后，CPU 继续执行刚才的程序。整个过程称为中断处理，也称为中断。

以上只是从宏观上描述中断从发生、处理到结束的过程，这对已经懂了的同学帮助不大，对于不懂的同学，仅仅是获得了浅表的认识。是的，我这样描述中断还远远不够。

通俗地说，中断的意思是打断正在做的事，本节开头的逛超市属于中断中最美好的一种，不过你懂的，大部分的中断并不是那么美。

咱们平时工作中，经常被各种邮件、会议以及各种通信工具打断，基本上“正经事”都在时间碎片中完成。如果戳中了您的泪点也不要急着共鸣，哈哈，因为还可以再难过一点，有时候这种中断还是嵌套的。比如正在忙着干活，同一个部门的 PM 同学过来说赶紧开会，于是被“强拉硬拽”进会议室，开会过程中，突然另一个 PM 闯进来说，找你半天了，赶紧跟我先去开另外一个会，于是再次被“毫无人性”地拖进另外一个会议室。

这种“中断”，似乎非常影响咱们的工作效率，至少在一些同学的眼里，这种中断是效率极低的。至于效率高，还是低，这取决于您看问题的角度了，其实它只是影响了大家眼里做“正经事”的效率，这些“正经事”大多数是自己要完成的工作，而打断我们的各种事情，表面上看是别人的工作，但其实里面或多或少都有自己的工作，要不干吗不找别人开会，淡定淡定。虽然自己手中的工作被滞后了，但由于及时参与了会议，PM 同学的工作又能继续向下进行了，整个部门的业务都同步发展，这样就使整体的效率都有很大提升。不过话说回来，这种过多的“中断”，对您这个人的利用率还是蛮高的，我只能说，同是天涯沦落人啊。这里向战斗在一线的运维工程师说一声：兄弟们辛苦了。

通过上面的例子，我们了解到，个人抽出一些时间，可以使整体的工作效率提升，也就是说，中断虽然是打断的意思，但它恰恰却是提升整个系统利用率最有效的方式，没有之一，因为有了中断，系统才能并发运行。这里解释下并发和并行的区别：并发指的是单位时间内的累积工作量，比如每秒并发数是 100，这是指一秒内累积的请求量总和为 100 个请求，属于并发。并行是指真正同时进行的工作量，比如并行 100 个请求量是指任意瞬间都有 100 个请求在发生，所以单核 CPU 谈并发，多核 CPU 谈并行。刚说的这个开会的例子就是把整个部门当作一个系统，自己当成了 CPU，虽然整体工作效率得到提升了，但不可否认，一会儿做这个一会儿做那个，CPU 确实好累，悄悄地表达对 CPU 的感激。

不再拿生活举例子了，咱们说说具体的计算机吧。我们平时使用的计算机都只有一个 CPU，即使是服务器，其 CPU 数量也不是无限多的，我见过的最多是 24 个核心。如果只有一个 CPU 的话，任何程序都必须以串行的方式、轮流调度到 CPU 上运行。不过话又说回来了，即使是有多个 CPU，如果任务在执行期间要求保证严格的

时序，或者说一个进程无法拆分成多个并行的部分（其实这是大部分程序的情况，很普遍），这种情况下该进程依然只会占用一个CPU，任意瞬间任务都只在一个核心上运行，只是根据CPU的负载情况，调度器会将该任务在多个CPU上挪来挪去，也就是说一会儿该任务在这个CPU上运行，一会儿可能就跑到另一个CPU上运行了，毫不夸张地说，如果系统中只有这一个任务的话，任意时间只有一个CPU在忙，其他几个核心还是空闲的。如果您对进程任意瞬间在哪个核心上运行感兴趣的话，在Linux中文件“/proc/进程pid/stat”第一行的第39个字段是“processor id”（可以通过man proc查看这方面的帮助），它表示进程本次执行时最后所在的CPU编号，编号是0～核心数-1，也就是进程在那一瞬间是在哪个CPU上执行，利用awk、perl等工具都可以直接用命令行的方式查看该字段，比较方便。如果还嫌麻烦的话还可以用ps命令查看，参数是：ps -eo pid，args，psr|grep 目标进程名。然而，在CPU外的设备是独立于CPU的，它与CPU是同步运行的，所以，CPU抽出一点时间来处理中断，外部设备就可以同CPU并行工作了，整个计算机系统可以大幅度地提升效率。

正是因为有了中断，我们才可以“同时”享受计算机的多种服务。

比如，现在的电影体积特别大，为节省空间，我们通常是把电影压缩转换成另外一种格式，这个转换时间是很长的。如果在转换过程中做不了其他的事情，用户是万万不能答应的，所以一边转换电影，我们还能一边敲键盘聊天，一边用鼠标点击网页。

现在清楚了，运用中断能够显著提升并发，从而大幅提升效率。

7.2 操作系统是中断驱动的

“没有中断，操作系统几乎什么都做不了，操作系统是中断驱动的”。这是我大学老师告诉我的。怎么理解这句话呢？

我个人是这样理解的，仅供参考：首先，操作系统是个死循环。不知是哪位大神说的，总之这句话说得太精辟了。其实这一点也不奇怪，如果操作系统不是死循环的话，简直不敢想像CPU会执行到哪里去，这就像咱们之前的代码中，如果代码结尾没有jmp $或while（1），CPU就成脱缰野马一样再也回不来了。它也许会把最后一条指令后面的数据解码成某种指令，也许会因为不识别该指令解码失败而抛出UD异常，即未定义操作码、无效操作码（Undefined，Invalid Opcode）。当然，我们所说操作系统的“死循环”并不是咱们这样在最后加个死循环代码，而是类似这样的形式。

```
while（1）{
    操作系统代码
}
```

其实，这个死循环本身做不了什么大事，仅仅是保证操作系统能够周而复始地运行下去，而运行的目的是为了等候某些事情发生。您看，我说的是等候，也就是操作系统是被动工作的，有事情发生它才会工作，所以它是被事件驱动的，而这个事件是以中断的形式通知操作系统的，所以说，操作系统是中断驱动的，这一点也没有错。

7.3 中断分类

大家已经了解，中断就是发生了某种事件需要通知CPU处理。所以，不管哪里，只要有事发生就应该让CPU知道。把中断按事件来源分类，来自CPU外部的中断就称为外部中断，来自CPU内部的中断称为内部中断。其实还可以再细分，外部中断按是否导致宕机来划分，可分为可屏蔽中断和不可屏蔽中断两种，而内部中断按中断是否正常来划分，可分为软中断和异常。

花点时间了解下基础的东西还是很有帮助的，以下咱们逐一介绍下这几种中断类型。

7.3.1 外部中断

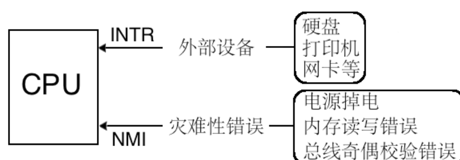
外部中断是指来自CPU外部的中断，而外部的中断源必须是某个硬件，所以外部中断又称为硬件中

断。比如说网卡收到了来自网络的数据包，这时候网卡就会主动通知 CPU，CPU 得到通知后便将数据拷贝到内核缓冲区。

为了让CPU获得每个外部设备的中断信号，最好的方式是在CPU中为每一个外设准备一个引脚接收中断，但这是不可能的，计算机中挂了很多外部设备，而且理论上外设数量是没有上限的，无论 CPU 中准备多少引脚都不够用，况且，我们还嫌 CPU 的体积太大呢，再整点引脚上去，CPU 岂不是更大了。所以一种可行的方案是 CPU 提供统一的接口作为中断信号的公共线路，所有来自外设的中断信号都共享公共线路连接到 CPU。

CPU 为大家提供了两条信号线。外部硬件的中断是通过两根信号线通知 CPU 的，这两根信号线就是 INTR（INTeRrupt）和 NMI（Non Maskable Interrupt）。它们的示意如图 7-1 所示。

大家看图 7-1 中，CPU 中有两条接收中断的信号线：INTR 和 NMI。也就是说所有的外部设备的中断信息都走一根信号线。



▲图 7–1 外部中断类型

郑重声明，以下所讨论的内容都是基于单核 CPU 的，好啦，讨论开始^_^。

在 CPU 上运行的程序都是串行的，所有任务，包括中断处理程序都是一个接一个在 CPU 上运行的，所有任务都共享同一个 CPU，CPU 在各个任务间不断切换才实现了并发。既然是串行的，也就是说每次 CPU 只能处理一个任务，何必整两根中断信号线？大家有没有觉得其中一根是多余的？是这样的，任何任务都有轻重缓急，有的中断发不发两可，甚至都不叫个事，有的中断发出了就是摊上大事了。CPU 为了区分这两种中断类型，通过不同的引脚加以区分，同一种类型的中断共用同一根信号线进入 CPU，这样 CPU 就不需要在每次收到中断时再辨析是哪种类型了。

只要从 INTR 引脚收到的中断都是不影响系统运行的，可以随时处理，甚至 CPU 可以不处理，假装没看见，因为它不影响 CPU 运行。而只要从 NMI 引脚收到的中断，那基本上全是硬伤，CPU 都没有运行下去的必要了。所以 CPU 更喜欢来自 INTR 的消息，苦点累点都没关系，相对 NMI 来说，至少它全是好消息。

咱们先聊聊可屏蔽中断。

可屏蔽中断是通过 INTR 引脚进入 CPU 的，外部设备如硬盘、网卡等发出的中断都是可屏蔽中断。可屏蔽的意思是此外部设备发出的中断，CPU 可以不理会，因为它不会让系统宕机，所以可以通过 eflags 寄存器的 IF 位将所有这些外部设备的中断屏蔽。另外，这些设备都是接在某个中断代理设备的，通过该中断代理也可以单独屏蔽某个设备的中断，这是后话，后面会有详细介绍。

对于这类可屏蔽中断，CPU 可以选择不用理会，甚至，即使在理会后，也可以像 Linux 那样，把中断分为上半部和下半部分开处理。

既然都说到这了，咱们扩展一下，上半部和下半部是什么？

操作系统是中断驱动的，中断发生后会执行相应的中断处理程序，我们希望 CPU 中断响应的时间越短越好，这样便能响应更多设备的中断。但是中断处理程序还是需要完整执行的，不能光为了提高中断响应效率而只执行部分中断处理程序。于是，把中断处理程序分为上半部和下半部两部分，把中断处理程序中需要立即执行的部分（分分钟不能耽误的部分）划分到上半部，这部分是要限时执行的，所以通常情况下只完成中断应答或硬件复位等重要紧迫的工作。而中断处理程序中那些不紧急的部分则被推迟到下半部中去完成。由于中断处理程序的上半部是刻不容缓要执行的，所以上半部是在关中断不被打扰的情况下执行的。当上半部执行完成后就把中断打开了，下半部也属于中断处理程序，所以中断处理程序下半部则是在开中断的情况下执行的，如果有新的中断发生，原来这个旧中断的下半部就会被换下 CPU，先执行新的中断处理程序的上半部，等待线程调度机制为旧中断处理程序择一日期（就是指调度算法认为的某个恰当时机）后，再调度其上 CPU 完成其下半部的执行。

还是拿网卡举例子，网络中的数据通过网线到达网卡后，首先会被存储到网卡自己的缓冲区中，这个缓冲区容量不大（比起内存来说是非常小的），即使很大也有写满的那天，所以里面的数据必须立即被 CPU 拿走，否则由于网卡缓冲区中无空余空间，后续到来的数据只能丢掉。鉴于这个刻不容缓的理由，网卡会立即发中断通知 CPU：“数据到了，赶紧取走”，这话说得无比坚定，丝毫没有商量的意思，CPU 立即放

下手里的工作（其实并不是真地立即放下，怎么也得把当前正在执行的指令执行完，指令的执行必须是原子操作一气呵成，哪有执行一半指令的道理），马上执行网卡的中断处理程序，将网卡缓冲区中的数据拷贝到内核缓冲区中，至此，救火工作算是完成了，这就是所说的上半部。CPU 拿到网络数据后，处理数据的工作就不那么紧急了，它将在下半部中完成，这部分将在适当的时机被启动。

即使像上面所说的那么紧急的中断，CPU 也是可以置之不理的，因为它不会让 CPU 宕机。下面说点让 CPU 宕机的中断——不可屏蔽中断。

不可屏蔽中断是通过 NMI 引脚进入 CPU 的，它表示系统中发生了致命的错误，它等同于宣布：计算机的运行到此结束了。

哪些错误会如此致命呢？

图 7-1 中列出了三种常见原因，每一种都足矣将计算机击倒，比如说内存读写错误，内存是 CPU 的舞台，各种程序都需要被载入内存后才能有机会运行。所以当 CPU 发现读写内存这个刚性需求都无法满足时，CPU 可以宣布收摊儿了，今天就到这了，再继续下去也没什么意思，因为什么都做不了。

不可屏蔽中断为什么称为不可屏蔽？

因为该中断表示出的问题太大了，要宕机了，屏蔽不了，因为根本不能假装没看见，比如断电了，机器都没法运行了，难道电脑还要死扛一会并说："没事没事，I'm ok，don't worry"吗？不可屏蔽中断可以理解成"即将宕机"中断。所以，这里的不可屏蔽不是说"不可以、不建议屏蔽"，而是真地处理不了啦，要能处理的话倒是可以先息事宁人屏蔽它，但真心处理不了，马上要宕机了，还能假装什么事都没发生吗？您懂的，掩耳盗铃自欺欺人的下场……所以 eflags 寄存器中的 IF 位对其无效也是没办法的事，即使 IF 位对它有效又能怎么样呢？问题已经出了，中断只是通知 CPU 而已，不是说通知收不到问题就没有发生，宕机是不可避免的。

CPU 收到中断后，得知道发生了什么事情才能执行相应的处理办法。这是通过中断向量表或中断描述符表（中断向量表是实模式下的中断处理程序数组，在保护模式下已经被中断描述符表代替，在后面章节中会细说）来实现的，首先为每一种中断分配一个中断向量号，中断向量号就是一个整数，它就是中断向量表或中断描述符表中的索引下标，用来索引中断项。中断发起时，相应的中断向量号通过 NMI 或 INTR 引脚被传入 CPU，中断向量号是中断向量表或中断描述符表里中断项的下标，CPU 根据此中断向量号在中断向量表或中断描述符表中检索对应的中断处理程序并去执行。

可屏蔽中断并不会导致致命问题，它的数量是有限的，所以每一种中断源都可以获得一个中断向量号。而不可屏蔽中断引起的致命错误原因有很多，每一种都是硬伤，出现了基本上可以认为用软件解决不了，多数属于物理上的问题，只能找硬件工程师解决了。所以，既然用软件解决不了，而且每种原因对于软件工程师来说都意义不大，就没必要再细分原因，统统为导致宕机的各种原因分配一个中断向量号就足够了，所以不可屏蔽中断的中断向量号为 2。

外部中断介绍到此结束，现在介绍下内部中断。

7.3.2　内部中断

内部中断可分为软中断和异常。

软中断，就是由软件主动发起的中断，因为它来自于软件，所以称之为软中断。由于该中断是软件运行中主动发起的，所以它是主观上的，并不是客观上的某种内部错误。

以下是可以发起中断的指令。

- "int 8 位立即数"。这是我们以后常用的指令，我们要通过它进行系统调用，8 位立即数可表示 256 种中断，这与处理器所支持的中断数是相吻合的。
- "int3"。这可不是 int 空格 3，它们之间无间隙。int3 是调试断点指令，其所触发的中断向量号是 3，以后在中断和异常表中大家会看到。我们用 gdb 或 bochs 调试程序时，实际上就是调试器 fork 了一个子进程，子进程用于运行被调试的程序。调试器中经常要设置断点，其原理就是父进程修改了子进程的指令，将其用 int3 指令替换，从而子进程调用了 int3 指令触发中断。用此指令实现调试的原理是 int3 指令的机器码是 0xcc，断点本质上是指令的地址，调试器（父进程）将被调试进程（子进程）断点起始地址的第 1 个字节备份好之后，

在原地将该指令的第 1 字节修改为 0xcc。这样指令执行到断点处时，会去执行机器码为 0xcc 的 int3 指令，该指令会触发 3 号中断，从而会去执行 3 号中断对应的中断处理程序，由于中断处理程序在运行时也要用到寄存器，为了保存所调试进程的现场，该中断处理程序必须先将当前的寄存器和相关内存单元压栈保存（提醒，当前寄存器和相关内存都属于那个被调试的进程），用户在查看寄存器和变量时就是从栈中获取的。当恢复执行所调试的进程时，中断处理程序需要将之前备份的 1 字节还原至断点处，然后恢复各寄存器和内存单元的值，修改返回地址为断点地址，用 iret 指令退出中断，返回到用户进程继续执行。

- into。这是中断溢出指令，它所触发的中断向量号是 4。不过，能否引发 4 号中断是要看 eflags 标志寄存器中的 OF 位是否为 1，如果是 1 才会引发中断，否则该指令悄悄地什么都不做，低调得很。
- bound。这是检查数组索引越界指令，它可以触发 5 号中断，用于检查数组的索引下标是否在上下边界之内。该指令格式是“bound 16/32 位寄存器， 16/32 位内存”。目的操作数是用寄存器来存储的，其内容是待检测的数组下标值。源操作数是内存，其内容是数组下标的下边界和上边界。当执行 bound 指令时，若下标处于数组索引的范围之外，则会触发 5 号中断。
- ud2。未定义指令，这会触发第 6 号中断。该指令表示指令无效，CPU 无法识别。主动使用它发起中断，常用于软件测试中，无实际用途。

值得注意的是以上几种软中断指令，除第一种的“int 8 位立即数”之外，其他的几种又可以称为异常。因为它们既具备软中断的“主动”行为，又具备异常的“错误”结果。

下面说说异常，异常是另一种内部中断，是指令执行期间 CPU 内部产生的错误引起的。

由于是运行时错误，所以它不受标志寄存器 eflags 中的 IF 位影响，无法向用户隐瞒（因为运行不下去了，错误兜不住了）。

对于中断是否无视 eflags 中的 IF 位，可以这么理解：

（1）首先，只要是导致运行错误的中断类型都会无视 IF 位，不受 IF 位的管束，如 NMI、异常。

（2）其次，由于 int n 型的软中断用于实现系统调用功能，不能因为 IF 位为 0 就不顾用户请求，所以为了用户功能正常，软中断必须也无视 IF 位。

总结：只要中断关系到“正常”运行，就不受 IF 位影响。

另外，这里所说的运行错误，是说指令语法方面的错误。

举个例子，比如说在执行 DIV 和 IDIV 除法指令时，处理器发现分母为 0（除 0，通常是程序忘记为分母赋值或传给分母的参数有误导致的），除法中分母是不能为 0 的，这不符合除法要求，将引发 0 号异常（叫中断也行）。

还有，当处理器无法识别某个机器码时，就会发起 6 号中断（异常），这和主动用 ud2 指令发起的中断是一样的。不过大部分 6 号中断可不是程序主动发起的，真地是 CPU 不认得 cs:ip 所指向的指令了。比如我们在程序尾用到了很多的死循环指令 jmp $和 while(1)，把它们去掉后，很可能 CPU 就把内存中的垃圾当成了指令来解码，如果碰巧能解码为某个指令，CPU 还能往前走一步，否则解码失败时就会抛出无效操作码 6 号异常。

并不是所有的异常都很致命，按照轻重程度，可以分为以下三种。

（1）Fault，也称为故障。这种错误是可以被修复的一种类型，属于最轻的一种异常，它给软件一次“改过自新”的机会。当发生此类异常时 CPU 将机器状态恢复到异常之前的状态，之后调用中断处理程序时，CPU 将返回地址依然指向导致 fault 异常的那条指令。通常中断处理程序中会将此问题修复，待中断处理程序返回后便能重试。最典型的例子就是操作系统课程中所说的缺页异常 page fault，话说 Linux 的虚拟内存就是基于 page fault 的，这充分说明这种异常是极易被修复的，甚至是有益的。

（2）Trap，也称为陷阱，这一名称很形象地说明软件掉进了 CPU 设下的陷阱，导致停了下来。此异常通常用在调试中，比如 int3 指令便引发此类异常，为了让中断处理程序返回后能够继续向下执行，CPU 将中断处理程序的返回地址指向导致异常指令的下一个指令地址。

（3）Abort，也称为终止，从名字上看，这是最严重的异常类型，一旦出现，由于错误无法修复，程序将无法继续运行，操作系统为了自保，只能将此程序从进程表中去掉。导致此异常的错误通常是硬件错

误，或者某些系统数据结构出错。

某些异常会有单独的错误码，即 error code，进入中断时 CPU 会把它们压在栈中，这是在压入 eip 之后做的，以后会说到，目前先了解下就行。

刚才说得那么热闹，说了好多几种异常，几种中断，咱们下面看看它们的庐山真面目，真的是真面目啊，原汁原味纯英文，见表 7-1。

表 7-1　　异常与中断

Vector No.	Mnemonic	Description	Source	Type	Error code(Y|N)
0	#DE	Divide Error	DIV and IDIV instructions.	Fault	N
1	#DB	Debug	Any code or data reference.	Fault/Trap	N
2	/	NMI Interrupt	Non-maskable external interrupt.	Interrupt	N
3	#BP	Breakpoint	INT3 instruction.	Trap	N
4	#OF	Overflow	INTO instruction.	Trap	N
5	#BR	BOUND Range Exceeded	BOUND instruction.	Fault	N
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.1	Fault	N
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.	Fault	N
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.	Abort	Y(0)
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.2	Fault	N
10	#TS	Invalid TSS	Task switch or TSS access.	Fault	Y
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.	Fault	Y
12	#SS	Stack Segment Fault	Stack operations and SS register loads.	Fault	Y
13	#GP	General Protection	Any memory reference and other protection checks.	Fault	Y
14	#PF	Page Fault	Any memory reference.	Fault	Y
15		Reserved			
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.	Fault	N
17	#AC	Alignment Check	Any data reference in memory.3	Fault	Y(0)
18	#MC	Machine Check	Error codes (if any) and source are model dependent.4	Abort	N
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction5	Fault	N
20～31		Reserved			
32～255		Maskable Interrupts	External interrupt from INTR pin or INT n instruction.	Interrupt	

表中 Error code 字段中，如果值为 Y，表示相应中断会由 CPU 压入错误码。

之前说到的中断向量是什么？其实就是表 7-1 中第一列的 Vector No.，即中断向量号。您也看到了，它就是个整数，范围是 0～255。

中断机制的本质是来了一个中断信号后，调用相应的中断处理程序。所以，CPU 不管有多少种类型的中断，为了统一中断管理，把来自外部设备、内部指令的各种中断类型统统归结为一种管理方式，即为每个中断信号分配一个整数，用此整数作为中断的 ID，而这个整数就是所谓的中断向量，然后用此 ID 作为中断描述符表中的索引，这样就能找到对应的表项，进而从中找到对应的中断处理程序。

中断向量的作用和选择子类似，它们都用来在描述符表中索引一个描述符，只不过选择子用于在 GDT

或 LDT 中检索段描述符，而中断向量专用于中断描述符表，其中没有 RPL 字段，以后说到中断描述符表时大伙就清楚了。

异常和不可屏蔽中断的中断向量号是由 CPU 自动提供的，而来自外部设备的可屏蔽中断号是由中断代理提供的（咱们这里的中断代理是 8259A），软中断是由软件提供的。

内部中断到这儿就结束了，前面花了一定篇幅介绍了外部中断，咱们今后要使用的就是外部中断中的可屏蔽中断，以后大家就知道了。

7.4 中断描述符表

中断描述符表（Interrupt Descriptor Table，IDT）是保护模式下用于存储中断处理程序入口的表，当 CPU 接收一个中断时，需要用中断向量在此表中检索对应的描述符，在该描述符中找到中断处理程序的起始地址，然后执行中断处理程序。

一直以来，我们都是讨论保护模式下的中断，值得一提的是中断可不是只在保护模式下才有，在实模式下当然也有中断，操作系统自古以来就是中断驱动的，实模式下要是没有中断也就没有操作系统了。只不过实模式下用于存储中断处理程序入口的表叫中断向量表（Interrupt Vector Table，IVT）。好啦，我这只是提一句而已，咱们还是要把精力放到中断描述符表上。

中断描述符表中有什么呢？

有人肯定会觉得我是个逗比，中断描述符表中的当然都是中断描述符啦，让大家感到意外的是表中不仅仅有中断描述符，还可以有任务门描述符和陷阱门描述符。由于表中所有描述符都是记录一段程序的起始地址，相当于通向某段程序的“大门”，所以，中断描述符表中的描述符有自己的名称——门。

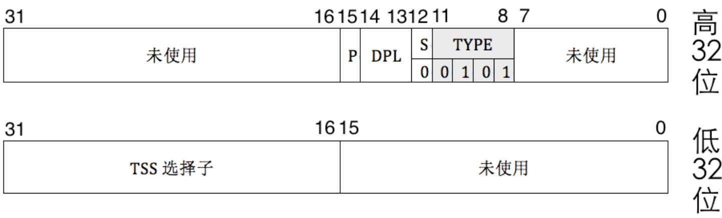
IDT 中只有这种称为门的描述符。之前在介绍特权级时介绍了门，这里给大伙复习一下。

门，顾名思义，是通往某处的入口。在计算机中，用门来表示一段程序的入口。拿它和段描述符对比一下就容易理解了，段描述符中描述的是一片内存区域，而门描述符中描述的是一段代码。

所有的描述符大小都是 8 字节，而门其实就是描述符，想通过门进入里面的世界是有条件的，就像娶媳妇“过门”一样，两个家庭得“门当户对”，这就是咱们人类社会中所说的门槛。这和之前咱们看到的段描述符功能类似，在门描述符中添加了各种属性，这就是进门的条件。当处理器把这些条件检查通过后就不再限制程序了。

之前在说段描述符的时候，记得曾经和大家说过，描述符高 4 字节的第 8～12 位是固定的意义，用来表述描述符的类型。CPU 通过这些位便知道该结构是哪种描述符。帮大伙儿回忆一下，第 8～11 位是 type 字段，用来描述一个描述符的类型，第 12 位是 S 位，用来表示系统段或非系统段（数据段）。S 为 0 时表示系统段，S 为 1 时表示数据段。type 字段是要和 S 字段配合在一起才能确定段描述符的确切类型。

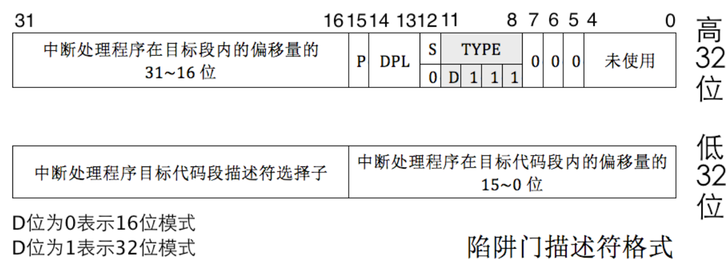
咱们这里的各种门都属于系统段，S 都等于 0，所以它们主要的区别就是 type 位不同。在第 5 章中已经介绍过四种门描述符了，不过那只涉及到有关特权级方面的内容，为方便大家，我把这四种门描述符再次贴到此处，下面再对它们详细介绍下。



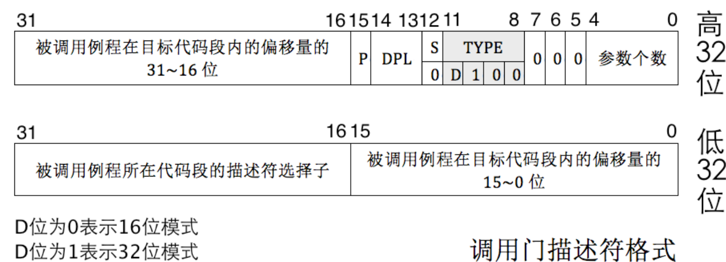
▲图 7-2　任务门描述符



▲图 7-3　中断门描述符



▲图 7-4　陷阱门描述符



▲图 7-5　调用门描述符

门描述符中的各属性位与段描述符中的属性意义相同，大伙可以参考全局描述符表部分的介绍。下面简要说下这几种门描述符，这里咱们只讨论 32 位保护模式。

1．任务门

任务门和任务状态段（Task Status Segment，TSS）是 Intel 处理器在硬件一级提供的任务切换机制，所以任务门需要和 TSS 配合在一起使用，在任务门中记录的是 TSS 选择子，偏移量未使用。任务门可以存在于全局描述符表 GDT、局部描述符表 LDT、中断描述符表 IDT 中。描述符中任务门的 type 值为二进制 0101，其结构如图 7-2 所示。顺便说一句大多数操作系统（包括 Linux）都未用 TSS 实现任务切换，咱们这里也不讨论啦。

2．中断门

中断门包含了中断处理程序所在段的段选择子和段内偏移地址。当通过此方式进入中断后，标志寄存器 eflags 中的 IF 位自动置 0，也就是在进入中断后，自动把中断关闭，避免中断嵌套。Linux 就是利用中断门实现的系统调用，就是那个著名的 int 0x80。中断门只允许存在于 IDT 中。描述符中中断门的 type 值为二进制 1110，其结构如图 7-3 所示。

3．陷阱门

陷阱门和中断门非常相似，区别是由陷阱门进入中断后，标志寄存器 eflags 中的 IF 位不会自动置 0。陷阱门只允许存在于 IDT 中。描述符中陷阱门的 type 值为二进制 1111。其结构如图 7-4 所示。

4．调用门

调用门是提供给用户进程进入特权 0 级的方式，其 DPL 为 3。调用门中记录例程的地址，它不能用 int 指令调用，只能用 call 和 jmp 指令。调用门可以安装在 GDT 和 LDT 中。描述符中调用门的 type 值为二进制 1100。其结构如图 7-5 所示。

除调用门外，另外的任务门、中断门、陷阱门都可以存在于中断描述符表中。

现代操作系统为了简化开发、提升性能和移植性等原因，很少用到调用门和任务门。所以本书中对它们的讨论也少之又少，我们会把精力放在中断门，像 Linux 那样，用它来实现系统调用。

一个中断源就会产生一个中断向量，每个中断向量都对应中断描述符表中的一个门描述符，任何中断源都通过中断向量对应到中断描述符表中的门描述符，通过该门描述符就找到了对应的中断处理程序。可见，中断发生后，采取什么样的动作是由中断处理程序决定的，但该程序是在中断描述符表中找到的，该表决定了中断信号落到哪个程序上，中断向量相当于子弹，门描述符相当于靶子，中断描述符表相当于狙击手，人家指哪就打哪，门描述符位置错了子弹就打错地方了，下面我们来看看这个威武霸气中断描述符表。

还记得很久很久以前说过的低端 1MB 内存布局吗？位于地址 0～0x3ff 的是中断向量表 IVT，它是实模式下用于存储中断处理程序入口的表。由于实模式下功能有限，运行机制比较“死板”，所以它的位置是固定的，必须位于最低端。大家看到了，已知 0～0x3ff 共 1024 个字节，又知 IVT 可容纳 256 个中断向量，所以每个中断向量用 4 字节描述。

对比中断向量表，中断描述符表有两个区别。

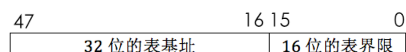
（1）中断描述符表地址不限制，在哪里都可以。

（2）中断描述符表中的每个描述符用 8 字节描述。

也许您心里有疑问，中断描述符表中可容纳多少个中断向量呢？咱们看看硬件是怎样支持的吧。

既然 IDT 的位置不固定，当中断发生时，CPU 是如何找到它的呢？

在 CPU 内部有个中断描述符表寄存器（Interrupt Descriptor Table Register，IDTR），该寄存器分为两部分：第 0～15 位是表界限，即 IDT 大小减 1，第 16～47 位是 IDT 的基地址，这和咱们之前介绍的 GDTR 是一样的原理。好啦，你懂啦，咱们的中断描述符表地址肯定要加载到这个寄存器中，只有寄存器 IDTR 指向了 IDT，当 CPU 接收到中断向量号时才能找到中断向量处理程序，这样中断系统才能正常运作。IDTR 结构如图 7-6 所示。



中断描述符表寄存器IDTR

▲图 7-6　IDTR 结构

16 位的表界限，表示最大范围是 0xffff，即 64KB。可容纳的描述符个数是 64KB/8=8K=8192 个。特别注意的是 GDT 中的第 0 个段描述符是不可用的，但 IDT 却无此限制，第 0 个门描述符也是可用的，中断向量号为 0 的中断是除法错。但处理器只支持 256 个中断，即 0～254，中断描述符中其余的描述符不可用。在门描述符中有个 P 位，所以，咱们将来在构建 IDT 时，记得把 P 位置 0，这样就表示门描述符中的中断处理程序不在内存中。

同加载 GDTR 一样，加载 IDTR 也有个专门的指令——lidt，其用法是：

lidt 48 位内存数据

在这 48 位内存数据中，前 16 位是 IDT 表界限，后 32 位是 IDT 线性基地址。

7.4.1　中断处理过程及保护

在了解中断描述符表之后，咱们说一下从中断发生到中断处理的过程，过程中涉及到特权级检查，也就是本节所说的保护。完整的中断过程分为 CPU 外和 CPU 内两部分。

CPU 外：外部设备的中断由中断代理芯片接收，处理后将该中断的中断向量号发送到 CPU。

CPU 内：CPU 执行该中断向量号对应的中断处理程序。

CPU 外这部分的处理过程，在后面咱们讲述中断代理芯片 Intel 8259A 时大家就会了解，这部分内容属于硬件范畴，我们这里只讨论处理器内的过程，这是软件能控制的部分，开始啦。

（1）处理器根据中断向量号定位中断门描述符。

中断向量号是中断描述符的索引，当处理器收到一个外部中断向量号后，它用此向量号在中断描述符表中查询对应的中断描述符，然后再去执行该中断描述符中的中断处理程序。由于中断描述符是 8 个字节，所以处理器用中断向量号乘以 8 后，再与 IDTR 中的中断描述符表地址相加，所求的地址之和便是该中断向量号对应的中断描述符。

（2）处理器进行特权级检查。

由于中断是通过中断向量号通知到处理器的，中断向量号只是个整数，其中并没有 RPL，所以在对由中断引起的特权级转移做特权级检查中，并不涉及到 RPL。中断门的特权检查同调用门类似，对于软件主动发起的软中断，当前特权级 CPL 必须在门描述符 DPL 和门中目标代码段 DPL 之间。这是为了防止位于 3 特权级下的用户程序主动调用某些只为内核服务的例程。

（a）如果是由软中断 int n、int3 和 into 引发的中断，这些是用户进程中主动发起的中断，由用户代码控制，处理器要检查当前特权级 CPL 和门描述符 DPL，这是检查进门的特权下限，如果 CPL 权限大于等于 DPL，即数值上 CPL≤门描述符 DPL，特权级“门槛”检查通过，进入下一步的“门框”检查。否则，处理器抛出异常。

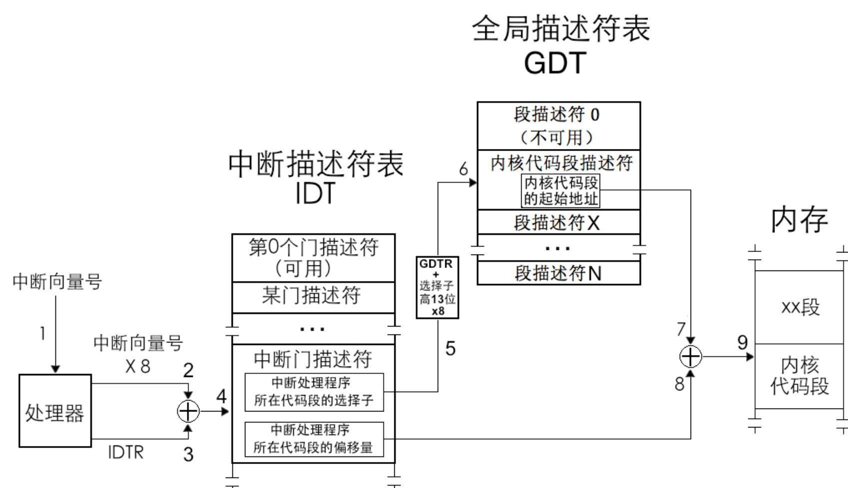
（b）这一步检查特权级的上限（门框）：处理器要检查当前特权级 CPL 和门描述符中所记录的选择子对应的目标代码段 DPL，如果 CPL 权限小于目标代码段 DPL，即数值上 CPL>目标代码段 DPL，检查通过。否则 CPL 若大于等于目标代码段 DPL，处理器将引发异常，也就是说，除了用返回指令从高特权级返回，特权转移只能发生在由低向高。

若中断是由外部设备和异常引起的，只直接检查 CPL 和目标代码段的 DPL，和上面的步骤 b）是一样的，要求 CPL 权限小于目标代码段 DPL，即数值上 CPL＞目标代码段 DPL，否则处理器引发异常。

（3）执行中断处理程序。

特权级检查通过后，将门描述符目标代码段选择子加载到代码段寄存器 CS 中，把门描述符中中断处理程序的偏移地址加载到 EIP，开始执行中断处理程序。

以上过程如图 7-7 所示。



▲图 7-7　中断处理过程

由于 IDT 中全都是门描述符，所以图 7-7 的 IDT 中的“某门描述符”表示中断门、陷阱门或任务门。

中断发生后，eflags 中的 NT 位和 TF 位会被置 0。如果中断对应的门描述符是中断门，标志寄存器 eflags 中的 IF 位被自动置 0，避免中断嵌套，即中断处理过程中又来了个新的中断，这是为防止在处理某个中断的过程中又来了个相同的中断，即同一种中断未处理完时又来了一个，这会导致一般保护性（GP）异常。这表示默认情况下，处理器会在无人打扰的方式下执行中断门描述符中的中断处理例程。

若中断发生时对应的描述符是任务门或陷阱门的话， CPU 是不会将 IF 位清 0 的。因为陷阱门主要用于调试，它允许 CPU 响应更高级别的中断，所以允许中断嵌套。而对任务门来说，这是执行一个新任务，任务都应该在开中断的情况下进行，否则就独占 CPU 资源，操作系统也会由多任务退化成单任务了。

从中断返回的指令是 iret，它从栈中弹出数据到寄存器 cs、eip、eflags 等，根据特权级是否改变，判断是否要恢复旧栈，也就是说是否将栈中位于 SS_old 和 ESP_old 位置的值弹出到寄存 ss 和 esp。当中断处理程序执行完成返回后，通过 iret 指令从栈中恢复 eflags 的内容。

当然如果大伙儿愿意的话，可以在中断处理程序中将 IF 位打开，这样便可以根据需要优先处理更高特权级的中断。可是如何打开此位呢？

有些 eflags 中的位需要以 pushf 压到栈中修改后再弹回的方式修改，这样有内存参与的操作必须是低效率，而且此操作是由多个步骤完成的，执行过程中容易被拆开，不能保证操作的原子性，原子性就是所有的步骤一次性全部完成，中间不能被打断，像原子那样不可再分（话说，数据库中的“事务”就是原子性的代表）。所以，处理器提供了专门用于控制 IF 位的指令，通过它，IF 可以直接控制。指令 cli 使 IF 位为 0，这称为关中断，指令 sti 使 IF 位为 1，这称为开中断。

IF 位只能限制外部设备的中断，对那些影响系统正常运行的中断都无效，如异常 exception，软中断，如 int n 等，不可屏蔽中断 NMI 都不受 IF 限制。

现在给大伙儿加餐了（如果已经觉得很撑了，可以先消化消化前面的内容再继续^^），前面说过，进入中断时要把 NT 位和 TF 位置为 0。TF 表示 Trap Flag，也就是陷阱标志位，这用在调试环境中，当 TF 为 0 时表示禁止单步执行，也就是说，进入中断后将 TF 置为 0，表示不允许中断处理程序单步执行，这是好理解的。至于为什么要把 NT 位置为 0 呢？这和中断返回指令 iret 有关，在这里简要和大伙儿说下，更多相关内容（TSS 和任务门）将在用户进程的章节中介绍。

NT 位表示 Nest Task Flag，即任务嵌套标志位，也就是用来标记任务嵌套调用的情况。任务嵌套调用是指 CPU 将当前正执行的旧任务挂起，转去执行另外的新任务，待新任务执行完后，CPU 再回到旧任务继续执行。为什么 CPU 执行完旧任务后还能回到新任务呢？原因是在执行新任务之前，CPU 做了两件准备工作。

（1）将旧任务 TSS 选择子写到了新任务 TSS 中的“上一个任务 TSS 的指针”字段中。

（2）将新任务标志寄存器 eflags 中的 NT 位置 1，表示新任务之所以能够执行，是因为有别的任务调用了它。

第（2）步中这个“别的任务”便是指 CPU 在第 1 步中写进新任务自己 TSS 的“上一个任务 TSS 的指针”字段中的值。

CPU 把新任务执行完后还是要回去继续执行旧任务的，怎样回到旧任务呢？这也是通过 iret 指令。iret 指令因此有了两个功能，一是从中断返回，另外一个就是返回到调用自己执行的那个旧任务，这也相当于执行一个任务。那么问题来了，对同一条 iret 指令，CPU 是如何知道该从中断返回呢，还是返回到旧任务继续执行呢？

这就用到 NT 位了，当 CPU 执行 iret 时，它会去检查 NT 位的值，如果 NT 位为 1，这说明当前任务是被嵌套执行的，因此会从自己 TSS 中“上一个任务 TSS 的指针”字段中获取旧任务，然后去执行该任务。如果 NT 位的值为 0，这表示当前是在中断处理环境下，于是就执行正常的中断退出流程。

7.4.2　中断发生时的压栈

我们只讨论 32 位保护模式下的中断情况。

中断在发生时，处理器收到一个中断向量号，根据此中断向量号在中断描述符表中找到相应的中断门描述符，门描述符中保存的是中断处理程序所在代码段的选择子及在段内偏移量，处理器从该描述符中加载目标代码段选择子到代码段寄存器 CS 及偏移量到指令指针寄存器 EIP。注意，由于 CS 加载了新的目标代码段选择子，处理器不管新的选择子和任何段寄存器（包括 CS）中当前的选择子是否相同，也不管这两个选择子是否指向当前相同的段，只要段寄存器被加载，段描述符缓冲寄存器就会被刷新，处理器都认为是换了一个段，属于段间转移，也就是远转移。所以，当前进程被中断打断后，为了从中断返回后能继续运行该进程，处理器自动把 CS 和 EIP 的当前值保存到中断处理程序使用的栈中。不同特权级别下处理器使用不同的栈，至于中断处理程序使用的是哪个栈，要视它当时所在的特权级别，因为中断是可以在任何特权级别下发生的。除了要保存 CS、EIP 外，还需要保存标志寄存器 EFLAGS，如果涉及到特权级变化，还要压入 SS 和 ESP 寄存器。

下面看看以上寄存器入栈情况及顺序，这里不再讨论有关特权检查的内容。

（1）处理器根据中断向量号找到对应的中断描述符后，拿 CPL 和中断门描述符中选择子对应的目标代码段的 DPL 比对，若 CPL 权限比 DPL 低，即数值上 CPL > DPL，这表示要向高特权级转移，需要切换到高特权级的栈。这也意味着当执行完中断处理程序后，若要正确返回到当前被中断的进程，同样需要将

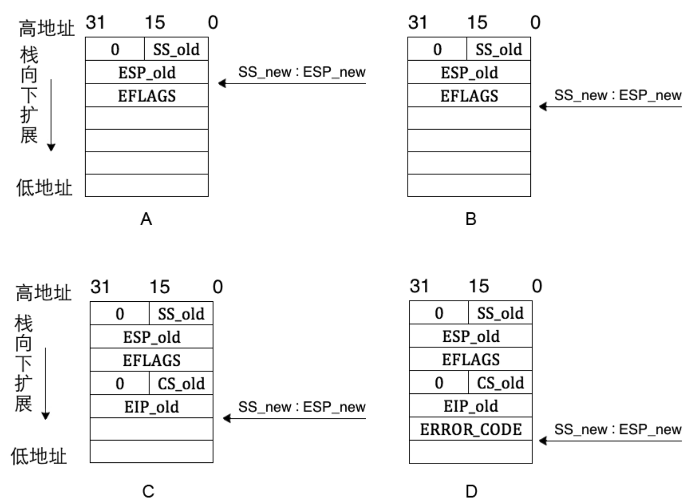
栈恢复为此时的旧栈。于是处理器先临时保存当前旧栈 SS 和 ESP 的值，记作 SS_old 和 ESP_old，然后在 TSS 中找到同目标代码段 DPL 级别相同的栈加载到寄存器 SS 和 ESP 中，记作 SS_new 和 ESP_new，再将之前临时保存的 SS_old 和 ESP_old 压入新栈备份，以备返回时重新加载到栈段寄存器 SS 和栈指针 ESP。由于 SS_old 是 16 位数据，32 位模式下的栈操作数是 32 位，所以将 SS_old 用 0 扩展其高 16 位，成为 32 位数据后入栈。此时新栈内容如图 7-8 中 A 所示。

（2）在新栈中压入 EFLAGS 寄存器，新栈内容如图 7-8 中 B 所示。

（3）由于要切换到目标代码段，对于这种段间转移，要将 CS 和 EIP 保存到当前栈中备份，记作 CS_old 和 EIP_old，以便中断程序执行结束后能恢复到被中断的进程。同样 CS_old 是 16 位数据，需要用 0 填充其高 16 位，扩展为 32 位数据后入栈。此时新栈内容如图 7-8 中 C 所示。当前栈是新栈，还是旧栈，取决于第 1 步中是否涉及到特权级转移。

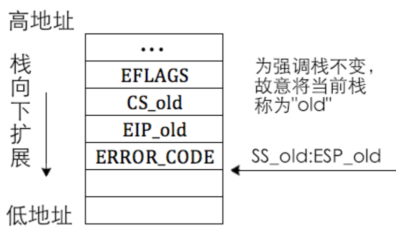
（4）某些异常会有错误码，此错误码用于报告异常是在哪个段上发生的，也就是异常发生的位置，所以错误码中包含选择子等信息，一会介绍。错误码会紧跟在 EIP 之后入栈，记作 ERROR_CODE。此时新栈内容如图 7-8 中 D 所示。

如果在第 1 步中判断未涉及到特权级转移，便不会到 TSS 中寻找新栈，而是继续使用当前旧栈，因此也谈不上恢复旧栈，此时中断发生时栈中数据不包括 SS_old 和 ESP_old。比如中断发生时当前正在运行的是内核程序，这是 0 特权级到 0 特权级，无特权级变化，如图 7-9 所示。



▲图 7-8　中断时特权级变化时的压栈

处理器进入中断执行完中断处理程序后，还要返回到被中断的进程，这是进入中断的逆过程。中断返回是用 iret 指令实现的。Iret，即 interrupt ret，此指令专用于从中断处理程序返回，假设在 32 位模式下，它从当前栈顶处依次弹出 32 位数据分别到寄存器 EIP、CS、EFLAGS。iret 指令并不清楚栈中数据的正确性，它只负责把栈顶处往上的数据，每次 4 字节，对号入座弹出到相关寄存器，所以在使用 iret 之前，一定要保证栈顶往上的数据是正确的，且从栈顶往上的顺序是 EIP、CS、EFLAGS，根据特权级是否有变化，还有 ESP、SS。



▲图 7-9　中断发生时无特权级变化时的栈中内容

由于段寄存器 CS 是 16 位，故从栈中返回的 32 位数据，其高 16 位被丢弃，只将低 16 位载入到 CS。若处理器发现返回后特权级会变化，还会继续将两个双字数据返回到 ESP、SS，其中 SS 也是 16 位寄存器，所以同样也是弹出 32 位数据后，只将其中的低 16 位加载到 SS。iret 指令意味着从中

断返回，所以，它是中断处理程序中最后一个指令。

顺便说一下，同类的指令还有 iretw 和 iretd，16 位模式下用 iretw，32 位模式下用 iretd。iret 是 iretw 和 iretd 的简写，无论是在 16 位模式，还是在 32 位模式下编码，都可以只用 iret 指令，它是被编译成 iretw，还是 iretd，取决于伪指令 BITS 所指明的字长。

iretw 隐含操作数是 16 位，所以只用在 16 位模式下。它依次从栈中分别弹出 2 字节到寄存器 IP、CS 和 eflags 中，在不涉及到特权级改变的情况下，栈指针 sp 自减 6。

iretd 隐含操作数是 32 位，所以只用在 32 位模式下。它先从栈中弹出 32 位数据到寄存器 EIP，再弹出 32 位数据，先丢弃高 16 位，只将低 16 位加载到 CS，再弹出 32 位数据到 eflags 中。在不涉及到特权级改变的情况下，栈指针 esp 自减 12。

注意，如果中断有错误码，处理器并不会主动跳过它的位置，咱们必须手动将其跳过，也就是说，在准备用 iret 指令返回时，当前栈指针 esp 必须指向栈中备份的 EIP_old 所在的位置，这样处理器才能将栈中的数据对号入座，弹出到各自对应的寄存器中。

在介绍调用门时和大家说过，处理器并不记得自己来到这儿之前（此处是指进入中断）已经做过了特权级检查，所以为了安全起见，处理器在返回到被中断过程中也要再进行一次特权级检查，下面咱们聊聊这个从中断处理程序返回的过程，假设此时已经手动将 ERROR_CODE 从栈中弹出，栈顶已位于正确的位置，即指向 EIP_old。

（1）当处理器执行到 iret 指令时，它知道要执行远返回，首先需要从栈中返回被中断进程的代码段选择子 CS_old 及指令指针 EIP_old。这时候它要进行特权级检查。先检查栈中 CS 选择子 CS_old，根据其 RPL 位，即未来的 CPL，判断在返回过程中是否要改变特权级。

（2）栈中 CS 选择子是 CS_old，根据 CS_old 对应的代码段的 DPL 及 CS_old 中的 RPL 做特权级检查，规则不再赘述。如果检查通过，随即需要更新寄存器 CS 和 EIP。由于 CS_old 在入栈时已经将高 16 位扩充为 0，现在是 32 位数据，段寄存器 CS 是 16 位，故处理器丢弃 CS_old 高 16 位，将低 16 位加载到 CS，将 EIP_old 加载到 EIP 寄存器，之后栈指针指向 EFLAGS。如果进入中断时未涉及特权级转移，此时栈指针是 ESP_old（说明在之前进入中断后，是继续使用旧栈）。否则栈指针是 ESP_new（说明在之前进入中断后用的是 TSS 中记录的新栈）。

（3）将栈中保存的 EFLAGS 弹出到标志寄存器 EFLAGS。如果在第 1 步中判断返回后要改变特权级，此时栈指针是 ESP_new，它指向栈中的 ESP_old。否则进入中断时属于平级转移，用的是旧栈，此时栈指针是 ESP_old，栈中已无因此次中断发生而入栈的数据，栈指针指向中断发生前的栈顶。

（4）如果在第 1 步中判断出返回时需要改变特权级，也就是说需要恢复旧栈，此时便需要将 ESP_old 和 SS_old 分别加载到寄存器 ESP 及 SS，丢弃寄存器 SS 和 ESP 中原有的 SS_new 和 ESP_new，同时进行特权级检查。补充，由于 SS_old 在入栈时已经由处理器将高 16 位填充为 0，现在是 32 位数据，所以在重新加载到栈段寄存器 SS 之前，需要将 SS_old 高 16 位剥离丢弃，只用其低 16 位加载 SS。

至此，处理器回到了被中断的那个进程。

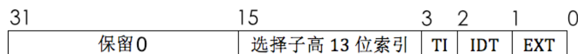
注意，如果在返回时需要改变特权级，将会检查数据段寄存器 DS、ES、FS 和 GS 的内容，如果在它们之中，某个寄存器中选择子所指向的数据段描述符的 DPL 权限比返回后的 CPL（CS.RPL）高，即数值上返回后的 CPL>数据段描述符的 DPL，处理器将把数值 0 填充到相应的段寄存器。原因在介绍调用门时说过啦，原理是选择子为 0 便指向 GDT 中第 0 个段描述符，该段描述符不可用，从而故意使处理器抛异常。如果大伙儿忘记啦，可以参考从调用门返回时的部分。

7.4.3　中断错误码

下面说说中断错误码。

有些中断会在栈中压入错误码，有点“临终遗言，提供线索”的意味，用来指明中断发生在哪个段上。所以，错误码最主要的部分就是选择子，只不过此选择子可以在多种表中检索描述符。错码码由几部分组成，格式如图 7-10 所示。

您看，错误码和选择子的格式很像吧，只是低 2 位不再是 RPL，而是 EXT 和 IDT。总之，错误码本质上就是个描述符选择子，通过低 3 位属性来修饰此选择子指向是哪个表中的哪个描述符。



▲图 7-10　错误码

EXT 表示 EXTernal event，即外部事件，用来指明中断源是否来自处理器外部，如果中断源是不可屏蔽中断 NMI 或外部设备，EXT 为 1，否则为 0。

IDT 表示选择子是否指向中断描述符表 IDT，IDT 位为 1，则表示此选择子指向中断描述符表，否则指向全局描述符表 GDT 或局部描述符表 LDT。

TI 和选择子中 TI 是一个意思，为 0 时用来指明选择子是从 GDT 中检索描述符，为 1 时是从 LDT 中检索描述符。当然，只有在 IDT 位为 0 时 TI 位才有意义。

选择子高 13 位索引就是选择子中用来在表中索引描述符用的下标。

高 16 位是保留位，全 0。

有时候不仅错误码的高 16 位全为 0，低 16 位也全为 0，那一个全 0 的错误码能指明什么信息？当全 0 的错误码出现时，表示中断的发生与特定的段无关，或者引用了一个空描述符，引用描述符就是往段寄存器中加载选择子的时候，处理器发现选择子指向的描述符是空的。

中断返回时，iret 指令并不会把错误码从栈中弹出，所以在中断处理程序中需要手动用栈指针跨过错误码或将其弹出。否则栈顶处若不是 EIP（EIP_old）的话，iret 返回时将会载入错误的值到后续寄存器。

通常能够压入错误码的中断属于中断向量号在 0～32 之内的异常，而外部中断（中断向量号在 32～255 之间）和 int 软中断并不会产生错误码。通常我们并不用处理错误码。

7.5　可编程中断控制器 8259A

我们本节将介绍可屏蔽中断的代理——可编程中断控制器 8259A。

8259A 的作用是负责所有来自外设的中断，其中就包括来自时钟的中断，以后我们要通过它完成进程调度。尽管我们实际对 8259A 编程的部分并不多，但不把它通透介绍一番的话，即使我告诉您应该如何使用它，您也不会真正清楚自己为什么要这么做，这必然不是本书的初衷。所以我一定要帮助大伙儿把 8259A 的工作原理搞清楚，我会努力的。

7.5.1　8259A 介绍

为了让 CPU 获得每个外部设备的中断信号，最好的方式是在 CPU 中为每一个外设准备一个引脚接收中断，但这是不可能的，计算机中挂了很多外部设备，而且外设数量是没有上限的，无论 CPU 中准备多少引脚都不够用，况且，引脚越多，体积越大，我们还嫌目前 CPU 的体积大呢。

上一节中我们说到，可屏蔽中断是通过 INTR 信号线进入 CPU 的，一般可独立运行的外部设备，如打印机、声卡等，其发出的中断都是可屏蔽中断，都共享这一根 INTR 信号线通知 CPU。大家想想看，任务是串行在 CPU 上执行的，CPU 每次只能执行一个任务，如果同时有多个外设发出中断，而 CPU 只能先处理一个，它到底先响应哪个呢？还有，为了不使这些中断丢失，是否要为它们单独维护一个中断队列？这些问题如果让 CPU 来做的话，似乎有些大材小用了，不仅要占用 CPU 时间，而且还要占用内存来存储中断队列，所以，干脆请个专人来做这事吧，这个专业人士就是中断代理，由它负责对所有中断仲裁，决定哪个中断优先被 CPU 受理。这有点像大臣和皇帝的关系，低阶官员写的奏折只能先交给大臣，由大臣决定是否要呈给皇帝过目。

中断代理有很多种，我们这里采用的是较流行的中断代理：Intel 8259A 芯片，就是本节所说的可编程中断控制器（PIC）8259A。

8259A 有哪些功能呢？

8259A 用于管理和控制可屏蔽中断，它表现在屏蔽外设中断，对它们实行优先级判决，向 CPU 提供中断向量号等功能。而它称为可编程的原因，就是可以通过编程的方式来设置以上的功能。

对于初学者，8259A 的设置有些复杂，咱们还是多花点篇幅来介绍，希望大伙儿不要嫌我啰嗦。

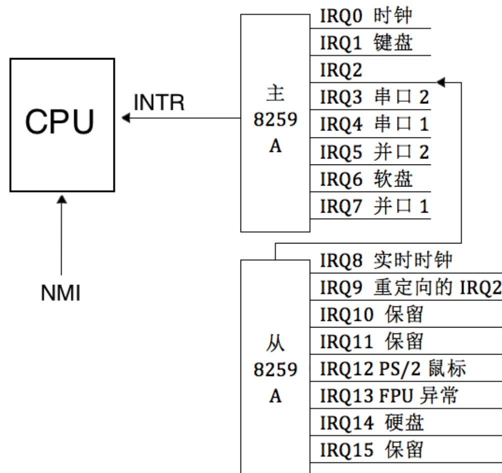
Intel 处理器共支持 256 个中断，但 8259A 只可以管理 8 个中断，不知道 Intel 这是要闹哪样，所以它为了多支持一些中断设备，提供了另一个解决方案，将多个 8259A 组合，官方术语就是级联。有了级联这种组合后，每一个 8259A 就被称为 1 片。若采用级联方式，即多片 8259A 芯片串连在一起，最多可级联 9 个，也就是最多支持 64 个中断。n 片 8259A 通过级联可支持 7n+1 个中断源，级联时只能有一片 8259A 为主片 master，其余的均为从片 slave。来自从片的中断只能传递给主片，再由主片向上传递给 CPU，也就是说只有主片才会向 CPU 发送 INT 中断信号。

每个独立运行的外部设备都是一个中断源，它们所发出的中断，只有接在中断请求（IRQ：Interrupt ReQuest）信号线上才能被 CPU 大神知晓，这也就是大家在开机时，电脑屏幕上会看到的 IRQ1…IRQn，这些都是为外部设备所分配的中断号。

我们说下什么是级联。由于单个 8259A 芯片只有 8 个中断请求信号线：IRQ0～IRQ7，这显然是不够用的，所以它提供了一种组合的方式，可以将多个自己像串联电路一样组合在一起，提供更多的中断请求信号线。这种组合方式就称为级联（cascade），乍一听是否觉得很熟悉？咱们平时用的交换机或 hub 都是这样做的，一个交换机上的网线接口是有限的，当上网的人多了接口不够用的时候，只能再买个交换机插在老交换机上做扩展，这就是典型的级联。8259A 也是这样的道理，无非是交换机传输的是数据链路层数据报文，8259A 传输的是外设的中断向量。

在咱们的个人电脑中只有两片 8259A 芯片，也就是说，一共 16 个 IRQ 接口。不过，单独使用哪个芯片都只能支持 8 个中断源，它们只有通过级联后才能都利用上，根据前面所说的公式，最多也只是支持 7*2+1=15 个中断。为什么不是 2*8=16 个？因为级联一个从片，要占用主片一个 IRQ 接口，而从片上的 IRQ 接口不被占用，从片上有专门的接口用于级联（相当于从片上向 CPU 发送 INT 信号的接口插在了主片上的某个 IRQ）。这和级联交换机的原理是一样的，交换机上通向网关的接口是单独的，下级交换机必须用该接口通过网线接在核心交换机的某个普通网卡接口上。

外部设备和 8259A 芯片是独立的，它们也得通过信号线连接到 8259A，主板电路上已经实现了这些，咱们只要把外设往主板上一插，这些设备自动就和 8259A 连接上了。这些设备在发中断的时候都以为是直接发给了 CPU，它们并不知道中间还隔着个中断代理，8259A 在收到了中断后，对中断判优，将优先级最高的中断转发给 CPU 处理。



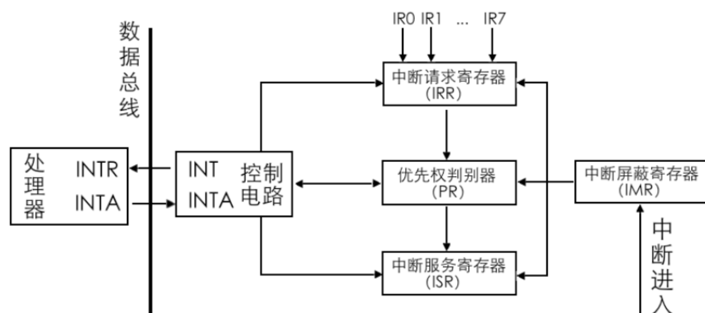
▲图 7-11　个人计算机中的级联 8259A

有了 8259A 这个中断代理，咱们重新审视下可屏蔽中断和 CPU 的关系，如图 7-11 所示。

8259A 内部的工作流程是怎么样的呢？为了给大家说清楚，咱们得了解下 8259A 的内部结构，请见图 7-12。

图 7-12 所示是 8259A 内部结构逻辑示意图，这里有一些信号和寄存器给大家介绍下。

- INT：8259A 选出优先级最高的中断请求后，发信号通知 CPU。
- INTA：INT Acknowledge，中断响应信号。位于 8259A 中的 INTA 接收来自 CPU 的 INTA 接口的中断响应信号。
- IMR：Interrupt Mask Register，中断屏蔽寄存器，宽度是 8 位，用来屏蔽某个外设的中断。
- IRR：Interrupt Request Register，中断请求寄存器，宽度是 8 位。它的作用是接受经过 IMR 寄存器过滤后的中断信号并锁存，此寄存器中全是等待处理的中断，“相当于”5259A 维护的未处理中断信号队列。
- PR：Priority Resolver，优先级仲裁器。当有多个中断同时发生，或当有新的中断请求进来时，将它与当前正在处理的中断进行比较，找出优先级更高的中断。
- ISR：In-Service Register，中断服务寄存器，宽度是 8 位。当某个中断正在被处理时，保存在此寄存器中。



▲图 7-12　8259A 芯片内部结构

以上介绍的寄存器都是 8 位，这是有意这样做的，其原因是 8259A 共 8 个 IRQ 接口，可以用 8 位寄存器中的每一位代表 8259A 的每个 IRQ 接口，类似于接口的位图，这样在后续的操作中，操作寄存器中的位便表示处理来自对应的 IRQ 接口的中断信号。

在“有图有真相”之后，咱们看看当 8259A 收到一个中断后会发生什么呢？现在可以介绍 8259A 的工作流程了。

当某个外设发出一个中断信号时，由于主板上已经将信号通路指向了 8259A 芯片的某个 IRQ 接口，所以该中断信号最终被送入了 8259A。8259A 首先检查 IMR 寄存器中是否已经屏蔽了来自该 IRQ 接口的中断信号。IMR 寄存器中的位，为 1，则表示中断屏蔽，为 0，则表示中断放行。如果该 IRQ 对应的相应位已经被置 1，即表示来自该 IRQ 接口上的中断已经被屏蔽了，则将该中断信号丢弃，否则，将其送入 IRR 寄存器，将该 IRQ 接口所在 IRR 寄存器中对应的 BIT 置 1。IRR 寄存器的作用“相当于”待处理中断队列。在某个恰当时机，优先级仲裁器 PR 会从 IRR 寄存器中挑选一个优先级最大的中断，此处的优先级决判很简单，就是 IRQ 接口号越低，优先级越大，所以 IRQ0 优先级最大。之后，8259A 会在控制电路中，通过 INT 接口向 CPU 发送 INTR 信号。信号被送入了 CPU 的 INTR 接口后，这样 CPU 便知道有新的中断到来了，又有活干了，于是 CPU 将手里的指令执行完后，马上通过自己的 INTA 接口向 8259A 的 INTA 接口回复一个中断响应信号，表示现在 CPU 我已准备好啦，8259A 你可以继续后面的工作。8259A 在收到这个信号后，立即将刚才选出来的优先级最大的中断在 ISR 寄存器中对应的 BIT 置 1，此寄存器表示当前正在处理的中断，同时要将该中断从“待处理中断队列”寄存器 IRR 中去掉，也就是在 IRR 中将该中断对应的 BIT 置 0。之后，CPU 将再次发送 INTA 信号给 8259A，这一次是想获取中断对应的中断向量号，就是我们前面所说的 0～255 的“整数”。由于大部分情况下 8259A 的起始中断向量号并不是 0（起始中断向量号被修改，原因后面会说），所以用起始中断向量号+IRQ 接口号便是该设备的中断向量号，由此可见，外部设备虽然会发中断信号，但它并不知道还有中断向量号这回事，不知道自己会被中断代理（如 8259A）分配一个这样的整数。随后，8259A 将此中断向量号通过系统数据总线发送给 CPU。CPU 从数据总线上拿到该中断向量号后，用它做中断向量表或中断描述符表中的索引，找到相应的中断处理程序并去执行。

处理流程到这就结束了吗？还早还早，这才刚完成了上半场。如果8259A 的“EOI 通知（End Of Interrupt）”若被设置为非自动模式（手工模式），中断处理程序结束处必须有向 8259A 发送 EOI 的代码，8259A 在收到 EOI 后，将当前正处理的中断在 ISR 寄存器中对应的 BIT 置 0。如果“EOI 通知”被设置为自动模式，在刚才 8259A 接收到第二个 INTA 信号后，也就是 CPU 向 8259A 要中断向量号的那个 INTA，8259A 会自动将此中断在 ISR 中对应的 BIT 置 0。

并不是进入了 ISR 后的中断就高枕无忧等着面见圣上 CPU 了，它还是有可能被后者换下来的。比如，在 8259A 发送中断向量号给 CPU 之前，这时候又来了新的中断，如果它的来源 IRQ 接口号比 ISR 中的低，也就是优先级更高，原来 ISR 中准备上 CPU 处理的旧中断，其对应的 BIT 就得清 0，同时将它所在的 IRR 中的相应 BIT 恢复为 1，随后在 ISR 中将此优先级更高的新中断对应的 BIT 置 1，然后将此新中断的中断向量号发给 CPU。您看，本来高高兴兴去面圣的，屁股还没坐热，结果还是被换了下来。当然，如果新来的中断优先级较低，依然会被放进 IRR 寄存器中等待处理。

以上整个过程就像皇帝上早朝一样，皇帝起床洗漱完毕后到了金銮殿（相当于 CPU 开机运行后），说了句："众位爱卿，有本启奏，无本退朝"。8259A 带着几本重要奏折说："老臣有本要奏"，用声音向皇上的耳朵发了个信号。皇上听到后回复一句："8259A，有何事啊？"也是用声音向 8259A 的耳朵传了个应答信号。8259A 一听，这表示皇上现在心情不错有时间处理奏折，于是把心里最重要的那个奏折挑了出来，剩下的几个奏折准备一会再启奏。这时候皇上说："呈上来"，于是 8259A 便把奏折交给了数据总线太监同学，皇上从太监那里拿到了奏折后，开始处理。

8259A 是"可编程"中断控制器，在了解 8259A 的工作原理后，咱们该去了解如何对其编程了。说实话，当初我头一次接触它的时候，觉得它的设置好麻烦，时隔今日，我依然觉得不顺手。对于新手来说确实有点难，不过咱们只做最基本的设置，能让它跑起来就行。

为叙述方便，咱们下面只讨论 32 位保护模式下的中断情况。

刚才咱们说过，外部设备不知道自己还有个中断向量号，这完全是 8259A 来分配的，其实这句话并不完全正确，因为 8259A 是咱们通过编程来设置的，归根结底还是咱们人来控制的，下面给各位说说，话要从头说起……

软件的舞台要靠硬件的支撑，为开发方便，很多功能都是由硬件原生支持的，因此，CPU 也提供了中断处理的框架。在此框架中，咱们只要填入所需要的数据即可，其他的工作由 CPU 自动运作。和中断处理相关的数据结构是中断描述符表和中断向量号，我们要做的工作就是准备这两项。中断描述符表也称为 IDT，我们会在下一节中介绍。既然称为"描述符"的表，说明它和全局描述符表 GDT 类似，表中的每一项都是 8 字节的描述符（这里还是要提一句，实模式下的中断向量表 IVT 中的每一项大小是 4 字节），我们曾经构建过 GDT，再次构建类似的结构一点也不难。不过有区别的是中断描述符表本质上就是中断处理程序地址数组，而中断向量号便是此数组的索引下标，这就是中断向量号是个整数的原因。CPU 不支持"数组名[索引]"的形式，那是高级语言编译器支持的东西，它最终也要编译转换成某种内存寻址方式之一，必须得用最基本的形式——地址来访问内存。当 CPU 接收到 8259A 送来的中断向量号后要将其乘以 8，再加上中断描述符表的起始地址，经过内存寻址，最终定位到目标中断处理程序。

以上说的是中断处理框架的流程，我们要做的确实很简单。

（1）构造好 IDT。

（2）提供中断向量号。

外部设备不知道中断向量号这回事，它只负责发中断信号。中断向量号是 8259A 传送给 CPU 的，而 8259A 是由咱们来控制的，中断描述符表也是咱们构造的，不知道大家有没有注意到，我们要做的事其实就是"自圆其说"，自己为外部设备设置好中断向量号，然后自己在中断描述符表中的对应项添加好合适的中断处理程序。

好啦，我不能再啰嗦了，本节到此结束，下节咱们看看如何通过编程 8259A 来设置这些。

7.5.2　8259A 的编程

既然 8259A 称为可编程中断控制器，就说明它的工作方式很多，咱们就要通过编程把它设置成需要的样子。对它的编程也很简单，就是对它进行初始化，设置主片与从片的级联方式，指定起始中断向量号以及设置各种工作模式。

其实，不光是咱们要操作 8259A，在开机之后的实模式下，BIOS 也对它光顾过，8259A 的 IRQ0～7 已经被 BIOS 分配了 0x8～0xf 的中断向量号。而在保护模式下，大家从表 7-1 中已经看到了，中断向量号为 0x8～0xf 的范围已经被 CPU 占了，分配给了各种异常，咱们还得重新为 8259A 芯片上的 IRQ 接口们分配中断向量号。

中断向量号是逻辑上的东西，它在物理上是 8259A 上的 IRQ 接口号。8259A 上 IRQ 号的排列顺序是固定的，但其对应的中断向量号是不固定的，这其实是一种由硬件到软件的映射，通过设置 8259A，可以将 IRQ 接口映射到不同的中断向量号。

在 8259A 内部有两组寄存器，一组是初始化命令寄存器组，用来保存初始化命令字（Initialization

Command Words，ICW），ICW 共 4 个，ICW1～ICW4。另一组寄存器是操作命令寄存器组，用来保存操作命令字（Operation Command Word，OCW），OCW 共 3 个，OCW1～OCW3。所以，我们对 8259A 的编程，也分为初始化和操作两部分。

- 一部分是用 ICW 做初始化，用来确定是否需要级联，设置起始中断向量号，设置中断结束模式。其编程就是往 8259A 的端口发送一系列 ICW。由于从一开始就要决定 8259A 的工作状态，所以要一次性写入很多设置，某些设置之间是具有关联、依赖性的，也许后面的某个设置会依赖前面某个 ICW 写入的设置，所以这部分要求严格的顺序，必须依次写入 ICW1、ICW2、ICW3、ICW4。
- 另一部分是用 OCW 来操作控制 8259A，前面所说的中断屏蔽和中断结束，就是通过往 8259A 端口发送 OCW 实现的。OCW 的发送顺序不固定，3 个之中先发送哪个都可以。

下面分别说一下每个 ICW 和 OCW，这是本节的重点，需要花点精力了，其实它们本身并不难，只是咱们接触得少，所以在学习过程中可能会显得不是那么轻松，建议先去洗个脸精神精神回来再看……闲话少说，哥几个走起。

ICW1 用来初始化 8259A 的连接方式和中断信号的触发方式。连接方式是指用单片工作，还是用多片级联工作，触发方式是指中断请求信号是电平触发，还是边沿触发。

注意，ICW1 需要写入到主片的 0x20 端口和从片的 0xA0 端口，如图 7-13 所示。

IC4 表示是否要写入 ICW4，这表示，并不是所有的 ICW 初始化控制字都需要用到。IC4 为 1 时表示需要在后面写入 ICW4，为 0 则不需要。注意，x86 系统 IC4 必须为 1。

ICW1

7	6	5	4	3	2	1	0
0	0	0	1	LTIM	ADI	SNGL	IC4

▲图 7-13　ICW1 格式

SNGL 表示 single，若 SNGL 为 1，表示单片，若 SNGL 为 0，表示级联（cascade）。这里说一下，若在级联模式下，这要涉及到主片（1 个）和从片（多个）用哪个 IRQ 接口互相连接的问题，所以当 SNGL 为 0 时，主片和从片也是需要 ICW3 的。

ADI 表示 call address interval，用来设置 8085 的调用时间间隔，x86 不需要设置。

LTIM 表示 level/edge triggered mode，用来设置中断检测方式，LTIM 为 0 表示边沿触发，LTIM 为 1 表示电平触发。

第 4 位的 1 是固定的，这是 ICW1 的标记，此时您可能不明白标记是什么，不过在本节的最后您将茅塞顿开。

第 5～7 位专用于 8085 处理器，x86 不需要，直接置为 0 即可。

ICW2 用来设置起始中断向量号，就是前面所说的硬件 IRQ 接口到逻辑中断向量号的映射。由于每个 8259A 芯片上的 IRQ 接口是顺序排列的，所以咱们这里的设置就是指定 IRQ0 映射到的中断向量号，其他 IRQ 接口对应的中断向量号会顺着自动排下去。

注意，ICW2 需要写入到主片的 0x21 端口和从片的 0xA1 端口如图 7-14 所示。

由于咱们只需要设置 IRQ0 的中断向量号，IRQ1～IRQ7 的中断向量号是 IRQ0 的顺延，所以，咱们只负责填写高 5 位 T3～T7，ID0～ID2 这低 3 位不用咱们负责。由于咱们只填写高 5 位，所以任意数字都是 8 的倍数，这个数字表示的便是设定的起始中断向量号。这是有意设计的，低 3 位能表示 8 个中断向量号，这由 8259A 根据 8 个 IRQ 接口的排列位次自行导入，IRQ0 的值是 000，IRQ1 的值是 001，IRQ2 的值便是 010……以此类推，这样高 5 位加低 3 位，便表示了任意一个 IRQ 接口实际分配的中断向量号。

ICW3 仅在级联的方式下才需要（如果 ICW1 中的 SNGL 为 0），用来设置主片和从片用哪个 IRQ 接口互连。


注意　ICW3 需要写入主片的 0x21 端口及从片的 0xA1 端口。

由于主片和从片的级联方式不一样，对于这个 ICW3，主片和从片都有自己不同的结构，如图 7-15 所示。

对于主片，ICW3 中置 1 的那一位对应的 IRQ 接口用于连接从片，若为 0 则表示接外部设备。比如，若主片 IRQ2 和 IRQ5 接有从片，则主片的 ICW3 为 00100100，如图 7-16 所示。

对于从片，要设置与主片 8259A 的连接方式，“不需要”指定用自己的哪个 IRQ 接口与主片连接，从

片上专门用于级联主片的接口并不是 IRQ。您想，如果从片用 IRQ 接口连接主片，若主片只级联一个从片，在从片上指定的 IRQ 接口便默认与主片上做级联的那个 IRQ 接口匹配了。但如果主片级联多个从片时，在从片上还要设置自己用于连接主片的 IRQ 接口与主片上连接从片的哪个 IRQ 接口（有多个）对接，这反而更麻烦。所以，设置从片连接主片的方法是只需要在从片上指定主片用于连接自己的那个 IRQ 接口就行了。在中断响应时，主片会发送与从片做级联的 IRQ 接口号，所有从片用自己的 ICW3 的低 3 位和它对比，若一致则认为是发给自己的。比如主片用 IRQ2 接口连接从片 A，用 IRQ5 接口连接从片 B，从片 A 的 ICW3 的值就应该设为 00000010，从片 B 的 ICW3 的值应该设为 00000101。所以，从片 ICW3 中的低 3 位 ID0～ID2 就够了，高 5 位不需要，为 0 即可。

ICW2

7	6	5	4	3	2	1	0
T7	T6	T5	T4	T3	ID2	ID1	ID0

▲图 7-14　ICW2

主片ICW3

7	6	5	4	3	2	1	0
S7	S6	S5	S4	S3	S2	S1	S0

▲图 7-15　主片 ICW3

ICW4 用于设置 8259A 的工作模式，当 ICW1 中的 IC4 为 1 时才需要 ICW4。

注意，ICW4 需要写入主片的 0x21 及从片的 0xA1 端口，如图 7-17 所示。

从片ICW3

7	6	5	4	3	2	1	0
0	0	0	0	0	ID2	ID1	ID0

▲图 7-16　从片 ICW3

ICW4

7	6	5	4	3	2	1	0
0	0	0	SFNM	BUF	M/S	AEOI	μPM

▲图 7-17　ICW4

ICW4 有些低位的选项基于高位，所以咱们从高位开始介绍。

第 7～5 位未定义，直接置为 0 即可。

SFNM 表示特殊全嵌套模式（Special Fully Nested Mode），若 SFNM 为 0，则表示全嵌套模式，若 SFNM 为 1，则表示特殊全嵌套模式。

BUF 表示本 8259A 芯片是否工作在缓冲模式。BUF 为 0，则工作非缓冲模式，BUF 为 1，则工作在缓冲模式。

当多个 8259A 级联时，如果工作在缓冲模式下，M/S 用来规定本 8259A 是主片，还是从片。若 M/S 为 1，则表示则表示是主片，若 M/S 为 0，则表示是从片。若工作在非缓冲模式（BUF 为 0）下，M/S 无效。

AEOI 表示自动结束中断（Auto End Of Interrupt），8259A 在收到中断结束信号时才能继续处理下一个中断，此项用来设置是否要让 8259A 自动把中断结束。若 AEOI 为 0，则表示非自动，即手动结束中断，咱们可以在中断处理程序中或主函数中手动向 8259A 的主、从片发送 EOI 信号。这种“操作”类命令，通过下面要介绍的 OCW 进行。若 AEOI 为 1，则表示自动结束中断。

μPM 表示微处理器类型（microprocessor），此项是为了兼容老处理器。若 μPM 为 0，则表示 8080 或 8085 处理器，若 μPM 为 1，则表示 x86 处理器。

4 个 ICW 都介绍过了，下面咱们看看用于操作 8259A 的各种 OCW 的格式。

OCW1 用来屏蔽连接在 8259A 上的外部设备的中断信号，实际上就是把 OCW1 写入了 IMR 寄存器。这里的屏蔽是说是否把来自外部设备的中断信号转发给 CPU。由于外部设备的中断都是可屏蔽中断，所以最终还是要受标志寄存器 eflags 中的 IF 位的管束，若 IF 为 0，可屏蔽中断全部被屏蔽，也就是说，在 IF 为 0 的情况下，即使 8259A 把外部设备的中断向量号发过来，CPU 也置之不理。

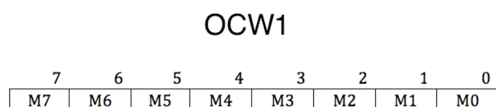
注意，OCW1 要写入主片的 0x21 或从片的 0xA1 端口，如图 7-18 所示。

M0～M7 对应 8259A 的 IRQ0～IRQ7，某位为 1，对应的 IRQ 上的中断信号就被屏蔽了。否则某位为 0 的话，对应的 IRQ 中断信号则被放行。

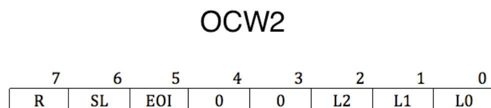
OCW2 用来设置中断结束方式和优先级模式。

注意，OCW2 要写入到主片的 0x20 及从片的 0xA0 端口。

OCW2 的配置比较复杂，各种属性位要配合在一起，组合出 8259A 的各种工作模式。如图 7-19 所示，由高 3 位 R、SL、EOI 可以定义多种中断结束方式和优先级循环方式。



▲图 7-18 OCW1



▲图 7-19 OCW2

在 OCW2 中比较灵活的是有个开关位：SL，可以针对某个特定优先级的中断进行操作，以下的优先级模式设置和中断结束都可以基于此开关做更细粒度的控制。

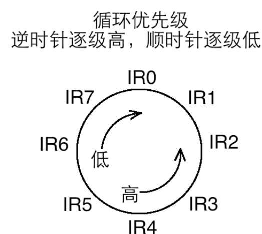
OCW2 其中的一个作用就是发 EOI 信号结束中断。如果使 SL 为 1，可以用 OCW2 的低 3 位（L2～L0）来指定位于 ISR 寄存器中的哪一个中断被终止，也就是结束来自哪个 IRQ 接口的中断信号。如果 SL 位为 0，L2～L0 便不起作用了，8259A 会自动将正在处理的中断结束，也就是把 ISR 寄存器中优先级最高的位清 0。

OCW2 另一个作用就是设置优先级控制方式，这是用 R 位（第 7 位）来设置的。

为表述方便，IRQ 各个接口在此被表述为“IR 数字”的形式，这也是微机接口中的命名规则。

如果 R 为 0，表示固定优先级方式，即 IRQ 接口号越低，优先级越高。

如果 R 为 1，表明用循环优先级方式，这样优先级会在 0～7 内循环。如果 SL 为 0，初始的优先级次序为 IR0>IR1>IR2>IR3>IR4>IR5>IR6>IR7。当某级别的中断被处理完成后，它的优先级别将变成最低，将最高优先级传给之前较之低一级别的中断请求，其他依次类推。所以，可循环方式多用于各中断源优先级相同的情况，优先级通过这种循环可以实现轮询处理。该循环可总结为如果 IR（i）优先级最低，IR（i+1）则优先级最高。其优先级关系如图 7-20 所示。



▲图 7-20 循环优先级

在图 7-20 中，顺时针方向的优先级是逐级减小，反之逆时针方向的优先级是逐渐增大。

比如，当前 IR3 为最高级别中断请求，处理完成后，IR3 将变成最低级别，IR4 变成最高级别，这一组循环之后的优先级变成了：

IR4>IR5>IR6>IR7>IR0>IR1>IR2>IR3

另外，还可以打开 SL 开关，使 SL 为 1，再通过 L2～L1 指定最低优先级是哪个 IRQ 接口。

举个例子，在 R 和 SL 都等于 1 的情况下，若想指定 IR5 为最低的优先级，需要将 L2～L0 置为 101。这样，参看图 7-20，新的初始优先级循环是：

IR6>IR7>IR0>IR1>IR2>IR3>IR4>IR5

以上就是 OCW2 的工作原理，咱们再细说一下各个属性位的意义，还从高位开始说起。

R,Rotation，表示是否按照循环方式设置中断优先级。R 为 1 表示优先级自动循环，R 为 0 表示不自动循环，采用固定优先级方式。

SL,Specific Level，表示是否指定优先等级。等级是用低 3 位来指定的。此处的 SL 只是开启低 3 位的开关，所以 SL 也表示低 3 位的 L2～L0 是否有效。SL 为 1 表示有效，SL 为 0 表示无效。

EOI，End Of Interrupt，为中断结束命令位。令 EOI 为 1，则会令 ISR 寄存器中的相应位清 0，也就是将当前处理的中断清掉，表示处理结束。向 8259A 主动发送 EOI 是手工结束中断的做法，所以，使用此命令有个前提，就是 ICW4 中的 AEOI 位为 0，非自动结束中断时才用。

值得注意的是在手动结束中断（AEOI 位为 0）的情况下，如果中断来自主片，只需要向主片发送 EOI 就行了，如果中断来自从片，除了向从片发送 EOI 以外，还要再向主片发送 EOI。

第 4～3 位的 00 是 OCW2 的标识。

L2～L0 用来确定优先级的编码，这里分两种，一种用于 EOI 时，表示被中断的优先级别，另一种用

于优先级循环时，指定起始最低的优先级别。

通过前面的介绍，其实整个 OCW2 就是各种关键字属性的配合使用，主要就是 L2～L0 需要配合 R 位、SL 位、EOI 位的设置。咱们看看这几种组合，见表 7-2。

表 7-2　　OCW2 高位属性组合

R	SL	EOI	描　述
0	0	1	普通 EOI 结束方式： 当中断处理完成后，向 8259A 发送 EOI 命令，8259A 会将 ISR 中当前级别最高的位置 0
0	1	1	特殊 EOI 结束方式： 当中断处理完成后，向 8259A 发送 EOI 命令，8259A 将 ISR 寄存器中由 L2～L0 指定的位清 0
1	0	1	普通 EOI 循环命令： 当中断处理完成后，8259A 将 ISR 中当前优先级最高的位清 0，并使此位的优先级变成最低，使原来第二高的优先级成为最高优先级。其他优先级类推，可以参照图 7-20
1	1	1	特殊 EOI 循环命令： 当中断处理完成后，8259A 将 ISR 中由 L2～L0 指定的相应位清 0，并使此位的优先级变成最低，使原来第二高的优先级成为最高优先级。其他优先级类推，可以参照图 7-20
0	0	0	清除自动 EOI 循环命令
1	0	0	设置自动 EOI 循环命令： 8259A 自动将 ISR 寄存器中当前处理的中断位清 0，并使此位的优先级变成最低，使原来第二高的优先级成为最高优先级。其他优先级类推，可以参照图 7-20
1	1	0	设置优先级命令： 将 L2～L0 指定的 IR(i)为最低优先级，IR(i+1)为最高优先级。其他优先级类推，可以参照图 7-20
0	1	0	无操作

到现在为止，咱们需要用到的部分已经说完了，还差一个 OCW3，虽然咱们用不上，但说实话也不差这一个了，还是捎带着说下。

OCW3 用来设定特殊屏蔽方式及查询方式，如图 7-21 所示。

注意，OCW3 要写入主片的 0x20 端口或从片的 0xA0 端口。

OCW3

7	6	5	4	3	2	1	0
/	ESMM	SMM	0	1	P	PR	RIS

▲图 7-21　OCW3

第 7 位未用到。

第 6 位的 ESMM（Enable Special Mask Mode）和第 5 位的 SMM（Special Mask Mode）是组合在一起用的，用来启用或禁用特殊屏蔽模式。ESMM 是特殊屏蔽模式允许位，是个开关。SMM 是特殊屏蔽模式位。只有在启用特殊屏蔽模式时，特殊屏蔽模式才有效。也就是若 ESMM 为 0，则 SMM 无效。若 ESMM 为 1，SMM 为 0，表示未工作在特殊屏蔽模式。若 ESMM 和 SMM 都为 1，这才正式工作在特殊屏蔽模式下。

第 4～3 位的 01 是 OCW3 的标识，8259A 通过这两位判断是哪个控制字。

P,Poll command，查询命令，当 P 为 1 时，设置 8259A 为中断查询方式，这样就可以通过读取寄存器，如 IRS，来查看当前的中断处理情况。

RR,Read Register，读取寄存器命令。它和 RIS 位是配合在一起使用的。当 RR 为 1 时才可以读取寄存器。

RIS,Read Interrupt register Select，读取中断寄存器选择位，顾名思义，就是用此位选择待读取的寄存器。有点类似显卡寄存器中的索引的意思。若 RIS 为 1，表示选择 ISR 寄存器，若 RIS 为 0，表示选择 IRR 寄存器。这两个寄存器能否读取，前提是 RR 的值为 1。

讲到这里，有关 8259A 的学习也就快到尾声了，但还有件事不吐不快，大家有没有疑惑，8259A 就 2 个端口地址，它是如何识别 4 个 ICW 和 3 个 OCW 的？

如果是初学者，我就当您有这个疑惑了。其实，这是有关内部寄存器寻址的问题，人家 8259A 有一套方法辨识自己。

ICW1 和 OCW2、OCW3 是用偶地址端口 0x20（主片）或 0xA0（从片）写入。

ICW2～ICW4 和 OCW1 是用奇地址端口 0x21（主片）或 0xA1（从片）写入。

以上 4 个 ICW 要保证一定的次序写入，所以 8259A 就知道写入端口的数据是什么了。

OCW 的写入与顺序无关，并且 ICW1 和 OCW2、OCW3 的写入端口是一致的，那 8259A 怎样来辨识它们呢？又是自问自答，其实就是各控制字中的第 4～3 标识位，通过这两位的组合来唯一确定某个控制字，见表 7-3。

表 7-3　控制字中标识汇总

控　制　字	第 4 位	第 3 位
ICW1	1	/
OCW2	0	0
OCW3	0	1

OCW1 是怎样确定的呢？OCW 是在初始化之后才有效的，所以在初始化之后写入奇地址端口的数据便被认为是 OCW1。

8259A 的编程就是写入 ICW 和 OCW，下面总结下写入的步骤。

对于 8259A 的初始化必须最先完成，步骤是：

- 无论 8259A 是否级联，ICW1 和 ICW2 是必须要有的，并且要顺序写入。
- 只有当 ICW1 中的 SNGL 位为 0 时，这表示级联，级联就需要设置主片和从片，这才需要在主片和从片中各写入 ICW3。注意，ICW3 的格式在主片和从片中是不同的。
- 只能当 ICW1 中的 IC4 为 1 时，才需要写入 ICW4。不过，x86 系统 IC4 必须为 1。

总结再总结，在 x86 系统中，对于初始化级联 8259A，4 个 ICW 都需要，初始化单片 8259A，ICW3 不要，其余全要。

在以上初始化 8259A 之后才可以用 OCW 对它操作。

好啦，到此有关 8259A 的介绍都结束了，大家有兴趣的话自行参阅相关书籍。

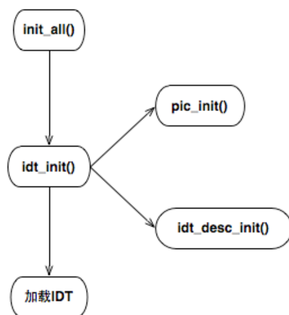
7.6 编写中断处理程序

为了实现中断处理，前面讲述了不少基础知识：特权级、内联汇编、中断描述符、中断代理 8259A，经过这些苦难之后，今天我们终于又开始写代码啦。这一节中，我们以循序渐进的方式，先写一个简陋的时钟中断，再逐渐丰富它，使其越来越“像模像样”。

7.6.1　从最简单的中断处理程序开始

本节我们将通过操作 8259A 打开中断，实现第一个中断处理程序。

Intel 8259A 芯片位于主板上的南桥芯片中，咱们不需要像网卡、硬盘那样单独安装才能用，也不需要为它的各个 IR 引脚指定连接的外部设备，这一切都安排好啦，比如主片 IR0 引脚上就是时钟中断，这已经由内部电路实现了，咱们只需要直接操作 8259A 就行，不用担心这些外部设备是否连接上了 8259A。



▲图 7-22　启用中断流程

程序初步计划以图 7-22 所示的流程开启中断，先和大家描述下愿景，这些还没开始做呢。

我们从上往下看，init_all 函数用来初始化所有的设备及数据结构，我们打算在主函数中调用它来完成初始化工作。

init_all 首先调用 idt_init，它用来初始化中断相关的内容。

由于初始化也要分成几部分来做，这由 pic_init 和 ide_desc_init 分别完成，idt_init 中调用这两个函数完成初始化工作。其中，pic_init 用来初始化可编程中断控制器 8259A。如果您在想为什么不叫 8259A_init，我解释一下，

PIC 就是可编程中断控制器 Programmable Interrupt Controller 的简称，而 8259A 也是 PIC 的一种，万一哪天想尝试用别的中断代理，函数名上就不用再改动了，其实就是为了扩展。

在用 pic_init 函数初始化 8259A 后，我们还需要初始化中断描述符表 IDT，这是用 ide_desc_init 来完成的。

在 idt_init 完成之后便可以加载 IDT 啦，到此打开中断的条件便准备好了。

1．用汇编语言实现中断处理程序

在这个初始化过程中，最核心、最底层的便是 ide_desc_init，在该函数中我们要填充中断处理程序的地址到 IDT 中，所以我们在执行该函数之前，需要提前把所有中断处理函数准备好。中断处理函数用汇编语言，还是 C 语言实现呢？为了较容易地演示中断机制的原理，大伙儿包容一下，就先用汇编吧，之后咱们再用 C 语言完成个升级版。

由于代码中我们用到了新的内容——宏，先提前给大伙介绍下。

宏属于预处理指令，预处理指令是编译器为用户编码方便而提供的、仅被编译器中的预处理器支持的符号，并不是处理直接支持的指令，故属于伪指令。

预处理指令是指在编译前，编译器需要预先处理的指令，也就是在编译前先扫描一下代码，将一些编译器提供的伪指令展开替换成具体的语言符号后编译器才能识别，也就是说代码在预处理之后，其中的预处理指令（伪指令）全都会不见的。而完成预处理工作的软件通常称为预处理器，其实就是一个功能模块，伪指令的意义只能编译器中的预处理器知道。

宏，即 Macro，宏是用来代替重复性输入，是一段代码的模板。不同的编译器基本上都会提供这样的预处理指令。

在汇编中定义宏有多种方式，如果是定义单行的宏，可以用%define 指令来实现，这和 C 语言中的 define 用法一致，不多说啦。

如果是定义多行的宏，就要用%macro 来实现，这个简单说一下。

其定义方法是：

```
%macro 宏名字参数个数
  …
  宏代码体
  …
%endmacro
```

在宏定义头中包含了“宏名字”，这是调用宏时用的，后面的“参数个数”是告诉预处理器，此宏可以支持的参数个数。在“宏代码体”部分，如果想引用某个参数，就要用“%数字”的方式来引用，比如%1 就表示第 1 个参数，以此类推。

参数在哪里？哪个是第 1 个参数呢？和函数调用一样，这是由调用宏时参数传递的顺序决定的。宏调用的方式为：

宏名称以逗号分隔参数列表

参数列表中最左边的参数就是第 1 个参数，参数序号并不是从 0 起的。我们在实际调用宏的时候，传入参数顺序要与宏代码体中引用的参数协调好。

举个例子，比如以下定义了一个宏。

```
%macro mul_add3
 mov eax,%1
add eax,%2
add eax,%3
%endmacro
```

用此方式调用：mul_add 45，24，33，其中%1 是 45，%2 是 24，%3 是 33。

该说的都说了，下面上干货，给各位看官呈上汇编版本的中断处理程序，见代码 7-1，它定义在新的文件中，这就是 kernel.S。

代码 7-1 （project/c7/a/kernel/kernel.S）

```
 1 [bits 32]
 2 %define ERROR_CODE nop          ;若在相关的异常中 CPU 已经自动压入了
                                   ;错误码，为保持栈中格式统一，这里不做操作
 3 %define ZERO push 0             ;若在相关的异常中 CPU 没有压入错误码
                                   ;为了统一栈中格式，就手工压入一个 0
 4
 5 extern put_str;声明外部函数
 6
 7 section .data
 8 intr_str db "interrupt occur!", 0xa, 0
 9 global intr_entry_table
10 intr_entry_table:
11
12 %macro VECTOR 2
13 section .text
14 intr%1entry:                    ;每个中断处理程序都要压入中断向量号
                                   ;所以一个中断类型一个中断处理程序
                                   ;自己知道自己的中断向量号是多少
15      %2
16      push intr_str
17      call put_str
18      add esp,4                  ; 跳过参数
19
20      ;如果是从片上进入的中断，除了往从片上发送 EOI 外，还要往主片上发送 EOI
21      mov al,0x20                ;中断结束命令 EOI
22      out 0xa0,al                ;向从片发送
23      out 0x20,al                ;向主片发送
24
25      add esp,4                  ;跨过 error_code
26      iret                       ;从中断返回，32 位下等同指令 iretd
27
28 section .data
29      dd   intr%1entry           ;存储各个中断入口程序的地址
                                   ;形成 intr_entry_table 数组
30 %endmacro
31
32 VECTOR 0x00,ZERO
33 VECTOR 0x01,ZERO
34 VECTOR 0x02,ZERO
…略
62 VECTOR 0x1e,ERROR_CODE
63 VECTOR 0x1f,ZERO
64 VECTOR 0x20,ZERO
```

先说重点内容，代码 7-1 定义了 33 个中断处理程序。每个中断处理程序都一样，就是调用字符串打印函数 put_str 来打印字符串“interrupt occur!”，之后直接退出中断。

为什么定义 33 个中断处理程序呢？

原因是中断向量 0～19 为处理器内部固定的异常类型，20～31 是 Intel 保留的，忘记的话，大伙赶紧往回翻看表 7-1。所以咱们可用的中断向量号最低是 32，将来咱们在设置 8259A 的时候，会把 IR0 的中断向量号设置为 32（这是后话）。目前只为演示中断机制的原理，咱们就拿连接在主片 IR0 接口上外部设备——时钟，小试身手。

由于目前咱们的中断处理程序一样，这是重复性的工作，而且是由多行代码组成，所以用宏来定义它们是再合适不过的啦。接下来看看是怎么实现的。

第 12～30 行是用宏定义中断处理程序的地方。在第 12 行，用 macro 定义了名为 VECTOR 的宏，其接受 2 个参数。

哪两个参数呢？您一定想到了，看看宏是怎么调用的不就行了吗。所以，暂且止步于此，咱们不妨先看看 62 行和 63 行，这是调用宏 VECTOR 的两个地方，本例中从 32～64 行一共有 33 个调用 VECTOR 的地方，即预处理后，会有 33 个中断处理程序。

拿第 62 行的“VECTOR 0x1e，ERROR_CODE”来说，第 1 个参数 0x1e 是中断向量号，用来表示：本宏是为了此中断向量号而定义的中断处理程序，或者说这是本宏实现的中断处理程序对应的中断向量

号，总之将来咱们要把它装载到中断描述符表中以该中断向量号为索引的中断门描述符位置。第 2 个参数 ERROR_CODE 也是个宏，它定义在第 2 行“%define ERROR_CODE nop”，它的值为 nop。nop 是汇编指令，它表示 no operation，不操作，什么都不干。在此完全是占位充数用的，一会儿说占位是怎么回事。

第 63 行的“VECTOR 0x1f，ZERO”，第 1 个参数 0x1f 是中断向量号，第 2 个参数是 ZERO，它也是个宏，定义在第 3 行“%define ZERO push 0”，也就是说 ZERO 的值是 push 0，是“把 0 压入栈”这个操作。

大家想，为什么有的宏的参数是 ERROR_CODE，而有的宏的参数是 ZERO？是时候解释占位的事啦。

如果您看了代码中的注释，想必您也能猜到是怎么回事。是这样的，之前咱们在介绍错误码时说过，有的中断会产生错误码，用来指明中断是在哪个段上发生的，错误码会在进入中断后，处理器在栈中压入寄存器 EIP 之后压入。

为了表述清楚，大伙儿最好了解进入中断后，处理器会向栈中压入数据的情况，虽然前面讲过啦，但为了不那么抽象，这里还是再简单说下。

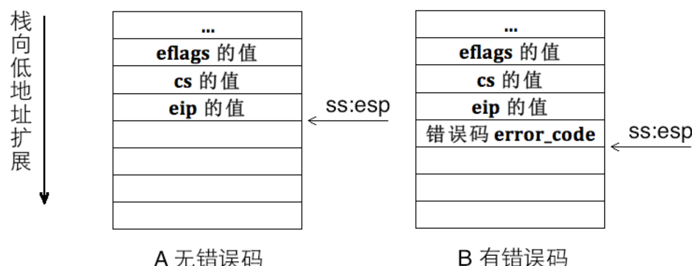
在中断发生时，处理器要在目标栈中保存被中断进程的部分寄存器环境，这是处理器自动完成的，不需要咱们手动写码。保存的寄存器名称及顺序是：

（1）如果发生了特权级转移，比如被中断的进程是 3 特权级，中断处理程序是 0 特权级，此时要把低特权级的栈段选择子 ss 及栈指针 esp 保存到栈中。

（2）压入标志寄存器 eflags。

（3）压入返回地址 cs 和 eip，先压入 cs，后压入 eip。

（4）如果此中断没有相应的错误码，至此，处理器把寄存器压栈的工作完成，栈中情况如图 7-23A 所示。如果此中断有错误码的话，处理器在压入 eip 之后会压入错误码，至此，处理器自动压栈工作全部完成，如图 7-23B 所示。



▲图 7-23　中断压入的栈

您看，有的中断会压入错误码，而有的中断则不会压入，说明这两种不同的中断发生时，即使栈顶初始值一样，由于个别的中断压入了错误码，最终栈指针也是不一样的，至少差了存储错误码的那 4 个字节。我们知道，用 iret 指令从中断返回时栈顶必须是 EIP 的值，也就是栈顶必须如图 7-23A 中 esp 指向的位置。所以如果栈中有错误码，在 iret 指令执行前必须要把栈中的错误码跨过。而目前的情况是不是所有的中断都有错误码，对于那些没有压入错误码的中断不需要跨过什么，难道要单独处理？可是我们是用宏来实现的，这是一个代码模板，具有通用性，怎么办呢？

大家也看出来了，问题的关键是栈顶指针不一致，即包含错误码的中断，其栈顶指针比不包含错误码的中断低了 4 字节。其实只要保证这两种情况的中断发生后，栈顶指针值是一样的就行了。让我们设想一下，我们只要保证在栈中 EIP 的位置之后还会再压入一个 32 位的数就成了，错误码是由处理器自动压入的，所以有错误码的中断咱们就不管了。如果没有错误码，咱们手工压入一个 32 位的数，这样不管中断是否会压入错误码，栈顶指针都是一样的，即栈结构相同，在 iret 指令执行前我们再将栈顶的数据（错误码或手工压入的 32 位数）跨过就万事大吉了。而做这件事的前提是，我们得知道哪些中断会压入错误码才行。

在表 7-1 中我们已经知道哪些中断会压入错误码，所以针对那些会压入错误码的中断，我们啥都不做，

如果中断未压入错误码，我们就手工压入个数。聪明的您绝对想到了，对，这就是我们把 ERROR_CODE 和 ZERO 作为参数的目的。

由于 ERROR_CODE 的值为 nop，所以对于那些参数为 ERROR_CODE 的宏调用，它们对应的都是包含错误码的中断。对于那些参数为 ZERO 的宏调用，它们对应的中断都不包括错误码，针对此种情况我们手工往栈中压个 0 占个位置，ZERO 其值为 push 0，这表示我们手工往栈中压入一个 0，这样后面在调用 iret 指令从中断返回之前，我们可以用通用的代码跨过栈顶的数据，即错误码或 0，以使栈顶指向 EIP，为 iret 准备好数据。

由于 EIP 和错误码都是处理器自动压入栈的，错误码是在 EIP 压入栈之后压入的，所以我们手工压入的 32 位数也应该放在压入 EIP 之后操作，即它必须是中断处理程序中的第一个指令。所以，大家看宏定义中的第一条指令是第 15 行的%2，它在预处理之后，会根据实际的参数展开为 nop 或 push 0，一会儿把预处理后的文件给大伙儿看下。

汇编中的 section，即节，用来定义一段相同属性的数据，该范围起始于当前 section 的定义处，一直持续到下一个 section 的定义处，若没有遇到新的 section 定义，则一直持续到文件结束处。

在这个宏中嵌套了两个 section，分别是第 13 行的.text，用作代码范围的起始定义，以及第 28 行的.data，一是用作上个.text 的结束，再有就是此数据范围的起始定义。由于我们调用了该宏 33 次，所以在预处理之后，宏中的这两个 section 将会各出现 33 次。

在.text 的 section 中，即代码的第 14～26 行，我们定义的是中断处理程序，这里大伙儿可能会对第 14 行的“intr%1entry：”有点疑问，这是在干吗？大家看，intr%1entry 的后面有个冒号，说明它是个标号，也就是地址，这是中断处理程序的起始处，所以此标号是为了获取中断处理程序的地址。由于我们目前有 33 个中断处理程序，标号不能重名，所以我们在 intr 和 entry 之间夹了个%1，上面看过宏调用的形式了，参数 1 是中断向量号，所以最终中断处理程序起始地址的范围是 intr[0～32]entry。

为了引用所有中断处理程序的地址，我们得事先把它们都记下来。为此，我们在 kernel.S 中定义了一个数组，数组名为 intr_entry_table，它在第 10 行定义，并且已经在第 9 行由 global 语句导出为全局符号，这样其他程序便可以使用此数组了。为了使此数组中的元素是每个中断处理程序的地址，我们在第 28～29 行定义了数据段（节），这就是 section .data 起的作用，由于 32 位下的地址是 4 字节，所以在第 29 行用伪指令 dd 来定义数组元素的宽度，元素值为 intr%1entry。这样每个宏调用都将在数组中产生新的地址元素。

也许您觉得有点不可思议，因为我们都知道，数组名是数组中所有数据元素的起始地址，数组元素所在的内存地址之间是连续的。而这里的数组名 intr_entry_table 定义在宏外，并且，其他数组元素，即中断程序地址，所在的 section 和数组名并不属于同一个 section 的定义，怎么就能保证数组元素地址是连续的呢？也就是说，它们是怎么能合并到一起了呢？

在很久很久之前就和大家说过啦，编译器会将属性相同的 section 合并到同一个大的 segment 中，而且，为了显得更靠谱一点，我们在 kernel.S 中对所有的数据 section 都用了同一个名字.data，这显得更万无一失啦。所以，大家放心，编译之后，所有中断处理程序的地址都会乖乖地作为数组 intr_entry_table 的元素紧凑地排在一起。

给大伙看一下预处理后的效果，这里用 nasm 提供的-E 参数即可，该参数告诉 nasm 仅做预处理，不编译，如图 7-24 所示。

在图 7-24 中的左右两部分其实是竖着排下来的，为了排版方便我才把下面的 0x20 号中断处理程序挪到了右边。

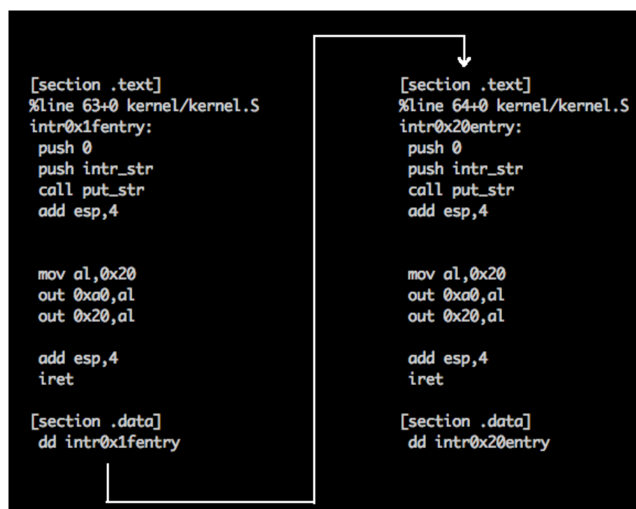
这只是其中的最后两个宏调用被替换后的代码，我们以 0x1f 和 0x20 为例，在 intr 和 entry 之间，它们已经被替换为实际的中断向量号 0x1f 和 0x20 啦。

以上罗哩罗嗦地介绍完了宏，咱们再看看其中的代码。

第 15 行的%是手工压入凑数 32 位数据，前面已有详述。

第 16～18 行是为了在中断程序中打印字符串“interrupt occur!”。参数 intr_str 定义在第 8 行，其值为字符串“interrupt occur!换行 0”的地址，结尾的 0 表示字符串结束。由于要调用 put_str，所以在第 5 行有 extern 声明了 put_str，它告诉 nasm，put_str 定义在别的文件中，链接时可以找到。第 18 行是跳过压入栈

中的参数，也就是 intr_str。



▲图 7-24　nasm 预处理

第 21～23 行是往主片和从片中写入 0x20，也就是写入 EOI。这是 8259A 的操作控制字 OCW2，其中第 5 位是 EOI 位，此位为 1，其余位全为 0，所以是 0x20。由于将来咱们在设置 8259A 时设置了手动结束，所以咱们得在中断处理程序中手动向 8259A 发送中断结束标记，否则 8259A 并不知道中断处理完成了，它会一直等下去，从而不再接受新的中断。也就是说，为了让 8259A 接受新的中断，必须要让 8259A 知道当前中断处理程序已经执行完成。

好啦，kernel.S 到这就说完啦。根据图 7-22 我们还有好多事要做呢。歇口气儿，下一节咱们继续。

2．创建中断描述符表 IDT，安装中断处理程序

上一节中完成了中断处理程序的编写，在本节中咱们要把它们安装到中断描述符表中。接下来再看看如何将这些中断处理程序地址装载到中断描述符中。上代码 7-2，这是我们新增的又一个文件，interrupt.c。

代码 7-2　（project/c7/a/kernel/interrupt.c）

```
 1 #include "interrupt.h"
 2 #include "stdint.h"
 3 #include "global.h"
…略
12 #define IDT_DESC_CNT 0x21         // 目前总共支持的中断数
13
14 /*中断门描述符结构体*/
15 struct gate_desc {
16     uint16_t   func_offset_low_word;
17     uint16_t   selector;
18     uint8_t    dcount;            //此项为双字计数字段，是门描述符中的第 4 字节
                                  //此项固定值，不用考虑
19     uint8_t    attribute;
20     uint16_t   func_offset_high_word;
21 };
22
23 // 静态函数声明，非必须
24 static void make_idt_desc(struct gate_desc* p_gdesc, uint8_t attr, intr_handler function);
25 static struct gate_desc idt[IDT_DESC_CNT];   // idt 是中断描述符表
                                             //本质上就是个中断门描述符数组
26
27 extern intr_handler intr_entry_table[IDT_DESC_CNT];     // 声明引用定义在 kernel.S
                                                        // 中的中断处理函数入口数组
28
    …略
50
51 /* 创建中断门描述符 */
```

```
52 static void make_idt_desc(struct gate_desc* p_gdesc, uint8_t attr, intr_handler function) {
53      p_gdesc->func_offset_low_word = (uint32_t)function & 0x0000FFFF;
54      p_gdesc->selector = SELECTOR_K_CODE;
55      p_gdesc->dcount = 0;
56      p_gdesc->attribute = attr;
57      p_gdesc->func_offset_high_word = ((uint32_t)function & 0xFFFF0000) >> 16;
58 }
59
60 /*初始化中断描述符表*/
61 static void idt_desc_init(void) {
62      int i;
63     for (i = 0; i < IDT_DESC_CNT; i++) {
64           make_idt_desc(&idt[i], IDT_DESC_ATTR_DPL0, intr_entry_table[i]);
65     }
66     put_str("   idt_desc_init done\n");
67 }
68
69 /*完成有关中断的所有初始化工作*/
70 void idt_init() {
71     put_str("idt_init start\n");
72     idt_desc_init();             // 初始化中断描述符表
73     pic_init();                  // 初始化 8259A
74
75     /* 加载 idt */
76     uint64_t idt_operand = ((sizeof(idt) - 1) | ((uint64_t)((uint32_t)idt << 16)));
77     asm volatile("lidt %0" : : "m" (idt_operand));
78     put_str("idt_init done\n");
79 }
80
```

见代码 7-2，这是文件 interrupt.c，我们有关中断的内容都在此文件中实现。

第 70 行的 idt_init 函数负责所有和中断相关的初始化工作，是中断初始化工作的主函数。

中断描述符 IDT 本质上就是中断门描述符的数组，门描述符的结构咱们也定义好啦，它是由文件顶端的 struct gate_desc 来描述的。

结构体中位置越偏下的成员，其地址越高。struct gate_desc 结构中成员的定义是参照中断门描述符来定义的，大伙参照中断门描述符的图自己对比一下，各成员名也一目了然，不多解释啦。描述符都是 8 字节，不信您自己算算 struct gate_desc 结构体中成员的大小，总和便是 8 字节。

有了门描述符后，便可以定义中断描述符表啦，在文件顶端的 static struct gate_desc idt[IDT_DESC_CNT]，就是我们定义的中断描述符表。您看，它的数据类型正是 struct gate_desc，数组大小是 IDT_DESC_CNT。IDT_DESC_CNT 定义在第 12 行，其值为 0x21，即 33，表示 33 个中断处理程序，也就是要定义 33 个中断门描述符。另外，由于 IDT 属于全局数据结构，所以我们声明它为 static 类型。

有了 IDT 之后，接下来的任务便是在每个中断门描述符中安装中断处理程序。

idt_desc_init 用来填充中断描述符表，这是重中之重。咱们深入进去看看是怎么回事。

idt_desc_init 定义在第 61 行，函数体中用了一个 for 循环，通过调用 make_idt_desc 函数在中断描述符表中创建了 IDT_DESC_CNT 个中断门描述符。

make_idt_desc 函数是用来创建中断门描述符的，它接受 3 个参数：中断门描述符的指针、中断描述符内的属性及中断描述符内对应的中断处理函数。

make_idt_desc 的原理是将后两个参数写入第 1 个参数所指向的中断门描述符中，实际上就是用后面的两个参数构建第 1 个参数指向的中断门描述符。它定义在 idt_desc_init 函数的上面，内部实现就是为结构体 structgate_desc 中的成员赋值。其中为选择子 selector 赋值的 SELECTOR_ K_CODE 定义在 global.h 中，它是个指向内核数据段的选择子。我觉得您可能不会对其中的赋值操作有疑惑，您可能感到疑惑的地方是这些属性的具体值是什么，比如想看下 global.h 中的内容，别急，一会儿看。参数是在调用时赋值的，因此咱们从调用入手。

make_idt_desc（&idt[i], IDT_DESC_ATTR_DPL0, intr_entry_table[i]），这是在函数 idt_desc_init 中调用的，第 1 个参数便是中断描述符表 idt 的数组成员指针，第 2 个参数 IDT_DESC_ATTR_DPL0 是描述符的属性，它同样定义在 global.h 中，别急一会儿给你看 global.h。第 3 个参数是在 kernel.S 中定义的中断描述符地址数组 intr_entry_table 中的元素值，即中断处理程序的地址。

值得一提的是 intr_entry_table 在文件开头是这样声明的：

extern intr_handler intr_entry_table[IDT_DESC_CNT]，其中有个数据类型是 intr_handler，这是我们在 interrupt.h 中自定义的类型，其定义为：

```
typedef void* intr_handler；
```

也就是说 intr_handler 是个空指针类型，该指针没有具体的类型，仅仅用来表示地址。这是因为 intr_handler 是用来修饰 intr_entry_table 的，intr_entry_table 中的元素都是普通地址。

由于我们在实际调用 make_idt_desc 的时候，传给它的指针是来自中断描述符表中的中断门描述符地址，即数组 idt 的某个数组元素指针，所以，通过这个循环把所有的中断描述符都填充好，这样我们将整个 IDT 准备好了。

现在把 global.h 搬出来给大伙看看，请见代码 7-3。

代码 7-3　（project/c7/a/kernel/global.h）

```
 1  #ifndef __KERNEL_GLOBAL_H
 2 #define __KERNEL_GLOBAL_H
 3 #include "stdint.h"
 4
 5 #define  RPL0  0
 6 #define  RPL1  1
 7 #define  RPL2  2
 8 #define  RPL3  3
 9
10 #define TI_GDT 0
11 #define TI_LDT 1
12
13 #define SELECTOR_K_CODE   ((1 << 3) + (TI_GDT << 2) + RPL0)
14 #define SELECTOR_K_DATA   ((2 << 3) + (TI_GDT << 2) + RPL0)
15 #define SELECTOR_K_STACK   SELECTOR_K_DATA
16 #define SELECTOR_K_GS     ((3 << 3) + (TI_GDT << 2) + RPL0)
17
18 /*--------------   IDT 描述符属性  ------------*/
19 #define  IDT_DESC_P      1
20 #define  IDT_DESC_DPL0   0
21 #define  IDT_DESC_DPL3   3
22 #define  IDT_DESC_32_TYPE     0xE  // 32 位的门
23 #define  IDT_DESC_16_TYPE     0x6  // 16 位的门,不会用到
                            // 定义它只为和 32 位门区分
24 #define  IDT_DESC_ATTR_DPL0 \
     ((IDT_DESC_P << 7) + (IDT_DESC_DPL0 << 5) + IDT_DESC_32_TYPE)
25 #define  IDT_DESC_ATTR_DPL3 \
     ((IDT_DESC_P << 7) + (IDT_DESC_DPL3 << 5) + IDT_DESC_32_TYPE)
26
27 #endif
```

刚才用到的两个宏分别在文件 global.h 的第 13 行和第 24 行。

好啦，关于装载中断描述符表的部分到这就完成了，本节到此结束，咱们离打开中断不远了，下面继续看其他部分。

3．用内联汇编实现端口 I/O 函数

和中断相关的数据准备好了，接下来只要把中断代理 8259A 设置好就可以啦。对 8259A 或任何硬件的控制都要通过端口，之前咱们在操作硬盘或显卡时，都是用 Intel 语法风格的汇编语言编写的，毕竟其使用方法不如 C 函数那样方便。工欲善其事，必先利其器，在进行下一步之前，咱们把常用的端口读写功能封装成 C 函数，这就用到了我们之前所学的内联汇编。

放心，代码不长，个个短小精干，我们把有关端口操作的函数定义在 io.h 中，这是我们又新增的一个文件，见代码 7-4。

代码 7-4　（project/c7/a/lib/kernel/io.h）

```
 1 /******************机器模式   *******************
 2       b -- 输出寄存器 QImode 名称，即寄存器中的最低 8 位:[a-d]l
```

```
  3      w -- 输出寄存器 HImode 名称，即寄存器中 2 个字节的部分,如[a-d]x
  4
  5      HImode
  6           "Half-Integer"模式，表示一个两字节的整数
  7      QImode
 8      "Quarter-Integer"模式，表示一个一字节的整数
  9 *****************************************************/
 10
 11 #ifndef __LIB_IO_H
 12 #define __LIB_IO_H
 13 #include "stdint.h"
 14
 15 /* 向端口 port 写入一个字节*/
 16 static inline void outb(uint16_t port, uint8_t data) {
 17 /*********************************************************
 18  对端口指定 N 表示 0～255, d 表示用 dx 存储端口号,
 19  %b0 表示对应 al,%w1 表示对应 dx */
 20   asm volatile ( "outb %b0, %w1" : : "a" (data), "Nd" (port));
 21 /******************************************************/
 22 }
 23
 24 /* 将 addr 处起始的 word_cnt 个字写入端口 port */
 25 static inline void outsw(uint16_t port, const void* addr, uint32_t word_cnt) {
 26 /*********************************************************
 27  +表示此限制即做输入,又做输出.
 28  outsw 是把 ds:esi 处的 16 位的内容写入 port 端口，我们在设置段描述符时,
 29 已经将 ds,es,ss 段的选择子都设置为相同的值了，此时不用担心数据错乱。*/
 30    asm volatile ("cld; rep outsw" : "+S" (addr), "+c" (word_cnt) : "d" (port));
 31 /******************************************************/
 32 }
33
 34 /* 将从端口 port 读入的一个字节返回 */
 35 static inline uint8_t inb(uint16_t port) {
 36      uint8_t data;
 37      asm volatile ("inb %w1, %b0" : "=a" (data) : "Nd" (port));
 38      return data;
 39 }
 40
 41 /* 将从端口 port 读入的 word_cnt 个字写入 addr */
 42 static inline void insw(uint16_t port, void* addr, uint32_t word_cnt) {
 43 /******************************************************
 44      insw 是将从端口 port 处读入的 16 位内容写入 es:edi 指向的内存,
 45      我们在设置段描述符时，已经将 ds,es,ss 段的选择子都设置为相同的值了,
 46      此时不用担心数据错乱。*/
 47      asm volatile ("cld; rep insw" : "+D" (addr), "+c" (word_cnt)
: "d" (port) : "memory");
 48 /******************************************************/
 49 }
 50
 51 #endif
```

文件 io.h 并不长，总共就 51 行，与其他 c 文件不同的是我们把函数的实现部分直接放到了以.h 为结尾的头文件中，这似乎显得有些怪异。

我们平时都是把函数体放在.c 文件中，头文件只用于存放函数声明，如果函数是全局作用域的话，链接后，外部文件就可以调用该函数了，所以，一般情况下，头文件都作为功能模块的接口而存在，头文件中对应的函数在程序中也只会存在一份。而我们的 io.h 却是函数的实现，并且，里面各函数的作用域都是 static，这表明该函数仅在本文件中有效，对外不可见。这意味着，凡是包含 io.h 的文件，都会获得一份 io.h 中所有函数的拷贝，也就是说同样功能的函数在程序中会存在多个副本，这样程序体积就会大一些。看上去哥们还是“清醒”的，但为什么还是干这样的“糊涂”事呢？

是这样的，这里的函数并不是普通的函数，它们都是对底层硬件端口直接操作的，通常由设备的驱动程序来调用，不用说，为了快速响应，函数调用上需要更加高效。而且，操作系统是为了让用户程序编写、执行更加方便才诞生的，归根结底是为了用户程序服务，所以它会让处理器的大多数时间花在 3 特权级的用户程序上。为了让处理器更多地为用户程序服务，操作系统（包括硬件驱动程序）必须减少自己占用处理器的时间，所以，对硬件端口的操作，只要求一个字——快。

但一般的函数调用需要涉及到现场保护及恢复现场，即函数调用前要把相关的栈、返回地址（CS 和 EIP）保存到栈中，函数执行完返回后再将它们从栈中恢复到寄存器。栈毕竟是内存，速度低很多，而且入栈、出栈这么多内存操作，对于想方设法提速的内核程序来说是无法忍受的。

因此，为了提速，在我们的实现中，函数的存储类型都是 static，并且加了 inline 关键字，它建议处理器将函数编译为内嵌的方式。内嵌函数大家都清楚吧，就是将所调用的函数体的内容，在该函数的调用处，原封不动地展开，这样编译后的代码中将不包含 call 指令，也就不属于函数调用了，而是顺次执行。虽然这会让程序大一些，但减少了函数调用及返回时的现场保护及恢复工作，提升了效率还是值得的。

好啦，解释清楚了，咱们简单看下里面定义了哪些函数。

io.h 中就定义了 4 个函数，分别是。

（1）一次写入 1 个字节的 outb 函数。

（2）一次写入多个字的 outsw 函数，注意，是以 2 字节为单位的。

（3）一次读入 1 个字节的 inb 函数。

（4）一次读入多个字的 insw 函数，同样以 2 字节为单位。

outb 函数接受两个参数，参数 port 是 16 位无符号整型的端口号，此类型可容纳 Intel 所支持的 65536 个端口号，参数 data 是 1 字节的整型数据，outb 的功能是将 data 中的 1 字节数据写入 port 所指向的端口。

函数实现是用内联汇编来实现的，内联汇编的格式是：

```
asm [volatile] (“assembly code” : output : input : clobber/modify):
```

按照以上格式，我们自己的代码是：

```
asm volatile ( "outb %b0, %w1" : : "a" (data), "Nd" (port));
```

首先必须要清楚，outb 指令格式为 outb %al，%dx，其中%al 是源操作数，指的是 8 位数据，%dx 是目的操作数，指的是数据所写入的端口。

我们要做的是通过 gcc 提供的各种约束和机器模式操作码将内联汇编形式凑成 outb %al，%dx，咱们一点点分析下是怎么做到的。

从右边的 input 开始看，形参变量 port 的约束有 2 个，分别是 N 和 d。N 为立即数约束，它表示 0～255 的立即数，也就是 8 位 1 字节所表示的范围，这样把写入的数据限制在 1 字节之内。d 为寄存器约束，它表示让 gcc 为 port 分配的寄存器可以是 dl、dx 或 edx。outb 的目的操作数是 dx，我们得想办法将寄存器明确为 dx。这可以用操作码 w 来实现，它表示使用寄存器的 HImode 名称，即寄存器中 2 字节的那个可独立使用的部分，也就是[a-d]x。操作码是跟序号占位符配合在一起来使用的，所以操作码 w 要随着序号占位符在内联汇编的“assembly code”中使用。大家看“assembly code”，port 所对应约束的序号占位符是%1，我们目的是使用 dx 寄存器，所以用%w1 来限制目的操作数为寄存器 dx。

继续说 output 中的 data 变量，它对应的约束为 a，这表示用寄存器 al、ax 或 eax 来存储该变量的值，前面说过 outb 指令的源操作数是寄存器 al，我们也必须将源操作数使用的寄存器明确为 al 才行。这可以通过机器模式操作码 b 来实现，操作码 b 表示寄存器的 QImode 部分，也就是寄存器中最低 8 位可独立使用的部分，即[a-d]l。同理，在“assembly code”中用%b0 表示 al 寄存器。

到现在为止，我们已经凑成了“outb %al，%dx”的形式，咱们的任务完成，接下来就是上处理器执行了。

其实，这几个函数实现都差不多，再者代码中的注释已经很详实了，大伙儿自己都能看懂，所以我再给大伙儿说一个。

说完了一个简单的写入端口函数，再介绍一个相对复杂一些的读入端口函数，insw。

insw 接受三个参数，无符号 16 位整型变量 port 是待读入的端口号，空指针变量 addr 是数据缓冲区，用于存储读出来的数据，无符号 32 位整型变量 word_cnt 是以字（2 字节）为单位的数据单位量。insw 函数的功能是将从端口 port 读入的 word_cnt 个字写入 addr。

insw 函数的核心是用同名汇编指令 insw 来实现的，该指令的功能是从端口 port 处读入的 16 位数据写入 es：edi 指向的内存，即一次读入 2 字节。猛地一看还真有点不解，还要指定段寄存器 es，咱们这是在

C 环境下，难道还要在内联汇编中为 es 重新赋选择子？

这一点大可不必担心，这时候平坦模型的好处就体现出来了。平坦模型下所有内存数据都在同一个 4GB 的大段中，即段基址为 0，段界限为 4GB-1。咱们在 loader 中初始化全局描述符表进入保护模式后，早已经把段寄存器 ds、es、ss 都指定为相同的数据段选择子啦（fs 未使用，所以未做初始化），忘记的话可见图 7-25。所以 es 和 ds 指向的是同一个基址为 0 的段，咱们只要指定偏移地址就行啦。平坦模型下的编译器，其所编译出来的程序中的符号地址自然按照平坦模型来编排，即默认段基址为 0，符号地址其实就是偏移地址。指针变量用来存储其他符号的地址，也就是说，此处指针变量 addr 中的值便是偏移地址，咱们只要把 addr 的值约束到 edi 寄存器就行了。于是，在 output 中，"+D" (addr)表示用寄存器约束 D 将变量 addr 的值约束到 EDI 中。

```
137 [bits 32]
138 p_mode_start:
139     mov ax, SELECTOR_DATA
140     mov ds, ax
141     mov es, ax
142     mov ss, ax
143     mov esp,LOADER_STACK_TOP
144     mov ax, SELECTOR_VIDEO
145     mov gs, ax
```

▲图 7-25　loader 中为段寄存器初始化

insw 函数实现的是将多个字从指定端口读出，而汇编指令 insw 执行一次只能从端口读取 2 字节的数据，所以这必然涉及到循环执行，这里用的是重复指令 rep，它把寄存器 ecx 作为循环计数器，每执行一次，ecx 的值就减 1，直到 ecx 为 0 时停止执行。这就很好理解了，"+c" (word_cnt)便是把 word_cnt 的值约束到寄存器 ecx 中作为循环次数。

+表示所修饰的约束既做输入（位于 input 中），也做输出（位于 output 中），也就是告诉 gcc 所约束的寄存器或内存先被读入，再被写入。

为什么要用+呢？原因是寄存器 edi 和 ecx 先被读入，又被写入啦，同时作为指令的输入和输出，这在之前咱们介绍内联汇编的修饰符'+'时已埋下了伏笔。

也许您隐约觉得，只有重复执行字符串操作指令才会这样，没错，让我们回忆下，曾经在第 5 章和大家介绍过数据复制的三剑客。

（1）字符串搬运指令族 movs[dw]

（2）重复执行指令 rep

（3）方向指令 cld 和 std

其实后两个是固定不变的，第 1 个字符串搬运指令可以替换为其他以 DS：[E]SI 作为数据源地址，以 ES：[E]DI 作为目的地址的字符串操作（传输）指令，这里的 insw 相当于 movsw。

提醒一下，movsw 用到了 esi 和 edi，所以这两个寄存器的值都会自动更新，但 insw 只用到了 edi，所以只会更新 edi，不会更新 esi。这在前面介绍内联汇编和数据复制“三剑客”时已强调过。

每执行一次 insw 指令，insw 要把 ES：EDI 作为数据缓冲区，这时 EDI 作为 insw 指令的输入。由于在 insw 执行前已经用 cld 指令清除了方向位 DF，故 insw 指令执行后，EDI 的值自动加 2，这时 EDI 作为输出。

rep 用 ecx 控制执行后面指令 insw 的次数，所以每次先要读取 ecx 的值判断是否为 0，这时 ecx 作为输入，执行完后，ecx 要减 1，这时 ecx 作为输出。

"d" (port)就不用说啦，就是把端口号 port 的值约束到 dx 寄存器中。

在内联汇编的 clobber/modify 部，这里还用到了内存破坏 memory，由于 insw 指令往内存 es：edi 处写入了数据，所以通知 gcc 这块内存已经改变。

不知大伙儿有没有疑问，edi 也被更新了啊，为什么 edi 不需要在 clobber/modify 中声明？其实之前已经说过，如果在 output 和 input 中通过寄存器约束指定了寄存器，gcc 必然会知道这些寄存器会被修改，不需要再重复通知啦。

好啦兄弟们，咱们就介绍到这，另外两个没介绍的函数我相信您绝对有能力掌握，大伙儿辛苦啦，今天到此为止。

4．设置 8259A

在准备好中断描述符后，接下来我们该对 8259A 编程啦。

前面 7.5 节中介绍过的 8259A 内容确实多了一些，但那是为了让大伙儿明白 8259A 从工作原理到对其编程是怎么回事，而我们这里的实际代码却不多，这就像摇控电视机，无非就那几个按钮按来按去，但

第一次接触电视机的时候，先得把十几页的说明书通读一遍后，才知道按下的那几个按钮是什么意思。在本节，只要您对 8259A 还有点印象，咱们就能顺利地读下去。

不过，估计大伙儿已经忘记怎么对 8259A 编程啦，因为我就经常忘。所以，大伙儿可以先看看前面对 8259A 的编程那一节的最后总结的部分，如果您懒得往回翻，这里给大家摘要了一些重点。

8259A 的编程就是写入 ICW 和 OCW，其中 ICW 是初始化控制字，共 4 个，ICW1～ICW4，用于初始化 8259A 的各个功能。OCW 是操作控制字，用于同初始化后的 8259A 进行操作命令交互。所以，对 8259A 的操作是在其初始化之后，对于 8259A 的初始化必须最先完成。

因为硬盘是接在了从片的引脚上，可参见图 7-12，将来实现文件系统是离不开硬盘的，所以我们这里使用的 8259A 要采用主、从片级联的方式。在 x86 系统中，对于初始化级联 8259A，4 个 ICW 都需要，必须严格按照 ICW1～4 顺序写入。

ICW1 和 OCW2、OCW3 是用偶地址端口 0x20（主片）或 0xA0（从片）写入。

ICW2～ICW4 和 OCW1 是用奇地址端口 0x21（主片）或 0xA1（从片）写入。

好啦，回忆结束，咱们现在真刀真枪上战场啦。大伙放心，此部分不像前面介绍它的时候那么冗长，此处对它的编程我们也只用了最基本的设置。不信的话大伙请见代码 7-5。

代码 7-5　（project/c7/a/kernel/interrupt.c）

```
…略
4 #include "io.h"
…略
7 #define PIC_M_CTRL 0x20           // 主片的控制端口是 0x20
8 #define PIC_M_DATA 0x21           // 主片的数据端口是 0x21
9 #define PIC_S_CTRL 0xa0           // 从片的控制端口是 0xa0
 10 #define PIC_S_DATA 0xa1         // 从片的数据端口是 0xa1
…略
29 /* 初始化可编程中断控制器 8259A */
 30 static void pic_init(void) {
 31
32     /*初始化主片 */
 33     outb (PIC_M_CTRL, 0x11);     // ICW1: 边沿触发,级联 8259, 需要 ICW4
 34     outb (PIC_M_DATA, 0x20);     // ICW2: 起始中断向量号为 0x20
                                    // 也就是 IR[0-7] 为 0x20 ~ 0x27
 35     outb (PIC_M_DATA, 0x04);     // ICW3: IR2 接从片
 36     outb (PIC_M_DATA, 0x01);     // ICW4: 8086 模式, 正常 EOI
 37
 38     /*初始化从片 */
 39     outb (PIC_S_CTRL, 0x11);     // ICW1: 边沿触发,级联 8259, 需要 ICW4
 40     outb (PIC_S_DATA, 0x28);     // ICW2: 起始中断向量号为 0x28
                                    // 也就是 IR[8-15]为 0x28 ~ 0x2F
 41     outb (PIC_S_DATA, 0x02);     // ICW3: 设置从片连接到主片的 IR2 引脚
 42     outb (PIC_S_DATA, 0x01);     // ICW4: 8086 模式, 正常 EOI
 43
 44     /*打开主片上 IR0,也就是目前只接受时钟产生的中断 */
 45     outb (PIC_M_DATA, 0xfe);
 46     outb (PIC_S_DATA, 0xff);
 47
 48     put_str("   pic_init done\n");
 49 }
…略
69 /*完成有关中断的所有初始化工作*/
 70 void idt_init() {
 71     put_str("idt_init start\n");
 72     idt_desc_init();             // 初始化中断描述符表
 73     pic_init();                  // 初始化 8259A
 74
 75     /* 加载 idt */
 76     uint64_t idt_operand = ((sizeof(idt) - 1) | ((uint64_t)((uint32_t)idt << 16)));
 77     asm volatile("lidt %0" : : "m" (idt_operand));
 78     put_str("idt_init done\n");
 79 }
 80
```

代码是不是很少？和之前 8259A 那么冗长的理论知识相比，此刻是否宽慰了很多？

代码 7-5 还是 interrupt.c，只是把设置 8259A 的代码摘了出来。在文件开头我们包含了之前编写的 io.h 文件，我们要用其中的 outb 函数来编程 8259A。

对于 8259A 的设置是用 pic_init 函数完成的，它在 idt_init 中被调用，其定义在第 30 行。

在 pic_init 中依次设置主片和从片 8259A，无论是主片，还是从片，都必须按顺序依次写入 ICW1、ICW2、ICW3、ICW4，这在前面介绍 8259A 编程时已经解释过啦。

为方便编码，我们在文件开头定义了四个宏来表示主、从片的控制端口和数据端口。详见宏定义处后的注释。

我们先设置主片，在第 33 行往主片中写入 ICW1，前面给大伙儿回忆过啦，ICW1 是往主、从片的偶数地址写入的，即主片端口地址是 0x20，从片端口地址是 0xA0。

通过刚刚出炉的 outb 函数往端口 PIC_M_CTRL（也就是主片的控制端口 0x20）写入 ICW1，其值为 0x11。对照图 7-14 ICW1 格式，第 0 位是 IC4 位，表示是否需要指定 ICW4，我们需要在 ICW4 中设置 EOI 为手动方式，所以需要 ICW4。由于我们要级联从片，所以将 ICW1 中的第 1 位 SNGL 置为 0，表示级联。设置第 3 位的 LTIM 为 0，表示边沿触发。ICW1 中第 4 位是固定为 1。其他位不设置。

第 34 行往主片中写入 ICW2，ICW2～ICW4 是写入主、从片的奇地址端口，即主片的 0x21 和从片的 0xA1，以下再讨论 ICW3～ICW4 时不再重复说明。

ICW2 专用于设置 8259A 的起始中断向量号，由于中断向量号 0～31 已经被占用或保留，从 32 起才可用，所以我们往主片 PIC_M_DATA 端口（主片的数据端口 0x21）写入的 ICW2 值为 0x20，即 32。这说明主片的起始中断向量号为 0x20，即 IR0 对应的中断向量号为 0x20，这是我们的时钟所接入的引脚。IR1～IR7 对应的中断向量号依次往下排。

第 35 行往主片中写入 ICW3。

ICW3 专用于设置主从级联时用到的引脚。我发现很多人在设置级联 8259A 时，都是用 IR2 引脚来级联从片的，所以我也是用 IR2 引脚作为主片级联从片的接口，我猜想其他引脚也行的，要不然 ICW3 为什么是 8 位的？不过我并未亲自测试，我们这里能保证 8259A 正常运行就可以，大家感兴趣的话可以自行测试。第 2 个引脚，即 IR2，在 ICW3 中将其置为 1，故 ICW3 值为 0x04，写入主片奇地址端口 0x21，即 PIC_M_DATA。

第 36 行往主片中写入 ICW4。

8259A 的很多工作模式都在 ICW4 中设置，可以参见图 7-18 ICW4 的格式，我们只要设置其中的第 0 位：μPM 位，它设置当前处理器的类型，咱们所在的开发平台是 x86，所以要将其置为 1。此外还要设置 ICW4 的第 1 位：EOI 的工作模式位。还记得 EOI 的作用吧，即 End OF Interrupt，就是告诉 8259A 中断处理程序执行完了，8259A 现在可以接受下一个中断信号啦。EOI 的工作模式位就是设置发送 EOI 的方式。如果为 1，8259A 会自动结束中断，这里我们需要手动向 8259A 发送中断，所以将此位设置为 0。其他位按默认就行了，所以 ICW4 的值为 0x01。写入主片奇地址端口 0x21，即 PIC_M_DATA。

至此，已经向主片发送了 4 个 ICW，接下来设置从片 8259A。

第 39 行向从片发送 ICW1，其意义和主片的 ICW1 一致，只不过是往从片的偶数地址端口 0xA0 发送，这是从片的控制端口，即 PIC_S_CTRL。

第 40 行向从片发送 ICW2，这是设置从片的起始中断向量号。由于主片的中断向量号是 0x20～0x27，故从片的中断向量号顺着它延续下来，从 0x28 开始，即 ICW2 值为 0x28，也就是 IR[8-15] 为 0x28 ～ 0x2F。ICW2 通过 outb 函数向从片的奇数地址端口 0xA1 写入，即 PIC_S_DATA。

第 41 行向从片发送 ICW3，ICW3 专用于设置级联的引脚，这里设置从片连接在主片的哪个 IRQ 引脚上。刚才在设置主片的时候是设置用 IR2 引脚来级联从片，所以此处要告诉从片连接到主片的 IR2 上，即 ICW2 值为 0x02，通过 outb 函数向从片的奇数地址端口 0xA1 写入，即 PIC_S_DATA。

第 42 行向从片发送 ICW4，同主片的 ICW4 一样，不再解释。

至此主、从片都已经初始化完成了。

我们立即就能测试中断处理程序啦，虽然我们在 kernel.S 中将所有的中断处理程序都用宏 macro 设置

成一样的了，但我们还是只测试一个中断源比较靠谱。为此，我们拿位于主片上的 IR0 引脚上的时钟中断举例，位于其他引脚的外部中断信号咱们统统屏蔽。

屏蔽某个外部设备中断信号可以通过设置位于 8259A 中的中断屏蔽寄存器 IMR 来实现，咱们只要将相应位置 1 就达到了屏蔽相应中断源信号的目的。顺便说一句，标志寄存器 eflags 中的 IF 位对所有外部中断有效，不能通过它来屏蔽某个外设的中断。

所以，现在还有一件事要做，就是设置中断屏蔽寄存器 IMR，只放行主片上 IR0 的时钟中断，屏蔽其他外部设备的中断。

第 45～46 行开始设置 IMR 寄存器只放行时钟中断。

这样的命令操作已经不属于初始化了，所以此时再向 8259A 发送的任何数据都称为操作控制字，即 OCW。往 IMR 寄存器中发送的命令控制字称为 OCW1，主片上的 OCW1 为 0xfe，即第 0 位为 0，表示不屏蔽 IR0 的时钟中断。其他位都是 1，表示都屏蔽。从片上的所有外设都屏蔽，所以发送的 OCW1 值为 0xff。OCW1 是写入主、从片的奇地址端口，即主片的 0x21 端口（PIC_M_DATA）和从片的 0xA1 端口（PIC_S_DATA）。

好啦，8259A 的设置到这就结束啦，下一节咱们将打开中断，让中断处理程序跑起来。

5．加载 IDT，开启中断

本节是开启中断的最后一个环节：把中断描述符表 IDT 的信息加载到 IDTR 寄存器。

可以参见图 7-6 IDTR 结构，IDTR 是 48 位的寄存器，低 16 位是 IDT 的界限，即 IDT 尺寸大小-1，高 32 位是 IDT 的线性基地址。

往 IDTR 中加载 IDT 的指令是 lidt，lidt 的操作数也要符合 IDTR 寄存器的结构，所以 lidt 的操作数也必须是 48 位，前 16 位是界限 limit，后 32 位是基址，只不过这 48 位的数据必须位于内存中，所以 lidt 的用法是：

lidt 48 位内存数据

代码 7-6　（project/c7/a/kernel/interrupt.c）

```
25 static struct gate_desc idt[IDT_DESC_CNT];
// idt 是中断描述符表，本质上就是个中断门描述符数组
…略
75     /* 加载 idt */
76     uint64_t idt_operand = ((sizeof(idt) - 1) | ((uint64_t)((uint32_t)idt << 16)));
77     asm volatile("lidt %0" : : "m" (idt_operand));
78     put_str("idt_init done\n");
79 }
80
```

见代码 7-6，我们的 IDT 定义在文件开头第 25 行，即数组 struct gate_desc idt。

由于 C 语言中没有 48 位的数据类型，所以我们用 64 位的变量 idt_operand 来代替，这是没问题的，lidt 中会取出 48 位数据做操作数，所以咱们只要保证 64 位变量中的前 48 位数据是正确的就行。为了给 lidt 凑出 48 位的操作数，在第 76 行。

（1）先用 sizeof(idt) – 1 得到 idt 的段界限 limit，这用作低 16 位的段界限。

（2）接下来再将 idt 的地址挪到高 32 位即可，这可以通过把 idt 地址左移 16 位的形式实现。由于数组名便是地址，即指针，故先将其转换成整数才能参与后面的左移运算。考虑到 32 位地址经过左移操作后，高位将被丢弃，万一原地址高 16 位不是 0，这样会造成数据错误，故需要将 idt 地址转换成 64 位整型后再进行左移操作，这样其高 32 位都是 0，经过左移操作依然能够保证其精度。由于指针只能转换成相同大小的整型，故 32 位的指针不能直接转换成 64 位的整型，所以采取迂回的作法，先将其转换成 uint32_t，再将其转换成 uint64_t，之后再对这个 64 位的无符号整型数据进行左移 16 位操作。这样 idt 地址被移到了 16～48 位，低 16 位自动填充为 0。

（3）之后再将以上两步的结果通过“按位或”运算符|组合到一起后，存储到变量 idt_operand 中。

虽然经过以上的三步得到的操作数是 64 位，但由于 lidt 的操作数是从内存地址处获得的，所以 lidt 依然只在该地址处（&idt_operand）取其中的 48 位数据当作操作数。

在第 77 行，通过内联汇编的形式将变量 idt_operand 通过内存约束 m 的形式传给 lidt 作为操作数，该

操作数对应的内存约束用序号占位符%0 表示，所以 lidt %0 便将 idt 载入了 IDTR 寄存器。

这里小扩展一下：不知道大伙儿是否还记得，内存约束是传递的 C 变量的指针给汇编指令当作操作数，也就是说，在“lidt %0”中，%0 其实是 idt_operand 的地址&idt_operand，并不是 idt_operand 的值。原因是 AT&T 语法的汇编语言把内存寻址放在最高级，任何数字都被看成是内存地址（所以立即数需要加前缀$表示），所以 lidt %0 直接便去%0 指向的内存地址处获取 48 位的操作数。而在 Intel 汇编语法中，立即数是最高级，也就是说数字就是数字，不代表地址，内存寻址并不是最高级，所以内存寻址需要用显式用中括号[]的方式，如 lidt [idt_ptr]。

好啦，数据都备齐了，一切就绪，就差按下启动按钮啦。

一直以来，我们都是自底向上为开启中断而准备数据的，做的工作就像是上菜之前的配菜，我们之前辛辛苦苦所做的工作得有人触发才行。回顾图 7-22 启用中断流程，我们只差 init_all 没做啦，它就是触发我们之前所有工作的按钮。

init_all 顾名思义，用来做所有初始化相关的工作，而中断初始化只是其中之一。鉴于这个原因，为将来扩展方便，我们单独写个文件用来调用所有模块的初始化主函数，就像调用 idt_init 一样。

这个比较简单，我们在 kernel 目录下再写个 init.c 文件，把 init_all 定义在其中就行啦，大伙儿见代码 7-7。

代码 7-7 （project/c7/a/ kernel/init.c）

```
1 #include "init.h"
2 #include "print.h"
3 #include "interrupt.h"
4
5 /*负责初始化所有模块 */
6 void init_all() {
7     put_str("init_all\n");
8     idt_init();    //初始化中断
9 }
```

是不是很简单？直接在 init_all 中调用 idt_init 就行啦。

init_all 是由 main.c 中的主函数 main 调用的，我们接下来修改 main.c，见代码 7-8。

代码 7-8 （project/c7/a/ kernel/main.c）

```
1 #include "print.h"
2 #include "init.h"
3 void main(void) {
4     put_str("I am kernel\n");
5     init_all();
6     asm volatile("sti");   //为演示中断处理,在此临时开中断
7     while(1);
8 }
```

为了在 main 中调用 init_all，main.c 文件开头处便包含了 init.h，里面就一句函数的声明：void init_all(void);。

真正到了开启中断的时刻了，为了让中断程序运行，我们得打开中断才行，打开中断是用 sti 指令，它将标志寄存器 eflags 中的 IF 位置 1，这样来自中断代理 8259A 的中断信号便被处理器受理啦。外部设备都是接在 8259A 的引脚上，由于我们在 8259A 中已经通过 IMR 寄存器将除时钟之外的所有外部设备中断都屏蔽了，这样开启中断后，处理器只会收到源源不断地时钟中断。

好，到这该说的都说啦，就差编译运行了。

为了目录不至于太乱，我建立了 build 目录，用于将所有目标文件和编译后的内核文件都放在此目录下。编译、链接、写入磁盘的步骤如下。

```
gcc -I lib/kernel/ -I lib/ -I kernel/ -c -fno-builtin \
          -o build/main.o kernel/main.c 回车

nasm -f elf -o build/print.o lib/kernel/print.S 回车

nasm -f elf -o build/kernel.o kernel/kernel.S 回车

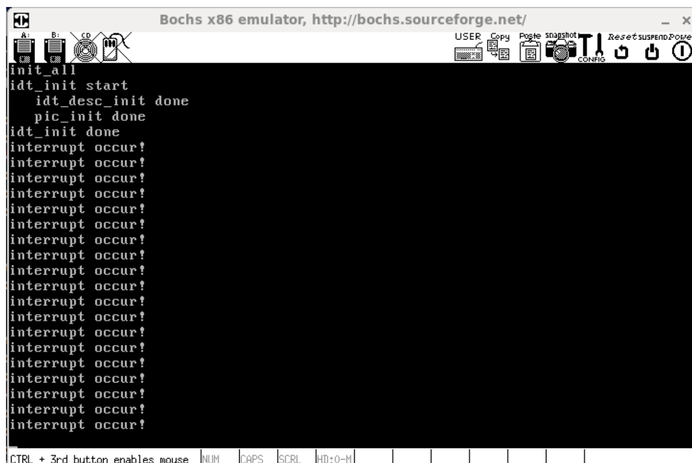
gcc -I lib/kernel/ -I lib/ -I kernel/ -c -fno-builtin\
```

```
          -o build/interrupt.o kernel/interrupt.c 回车

gcc -I lib/kernel/ -I lib/ -I kernel/ -c -fno-builtin \
          -o build/init.o kernel/init.c 回车
ld -Ttext 0xc0001500 -e main -o build/kernel.bin build/main.o build/init.o\
 build/interrupt.o build/print.o build/kernel.o 回车

dd if=build/kernel.bin of=/home/work/my_workspace/bochs/hd60M.img\
 bs=512 count=200 seek=9 conv=notrunc
记录了 12+1 的读入
记录了 12+1 的写出
6172 字节（6.2 kB）已复制，8.994e-05 秒，68.6 MB/s
```

进入 bochs 运行，结果如图 7-26 所示。



▲图 7-26　中断运行

看，每执行一个中断处理程序将会打印字符串“interrupt occur!”一次并换行，这里出现了多次中断，我们成功了。

有没有同学想看下安装后的中断描述符是什么样子？这个简单，只要我们在 bochs 中输入 info idt 就可以看到啦，下面给大伙儿截了张图，如图 7-27 所示。

中断处理程序地址

```
<bochs:10> info idt
Interrupt Descriptor Table (base=0xc0002cc0, limit=263):
IDT[0x00]=32-Bit Interrupt Gate target=0x0008:0xc0001880, DPL=0
IDT[0x01]=32-Bit Interrupt Gate target=0x0008:0xc0001899, DPL=0
IDT[0x02]=32-Bit Interrupt Gate target=0x0008:0xc00018b2, DPL=0
IDT[0x03]=32-Bit Interrupt Gate target=0x0008:0xc00018cb, DPL=0
IDT[0x04]=32-Bit Interrupt Gate target=0x0008:0xc00018e4, DPL=0
IDT[0x05]=32-Bit Interrupt Gate target=0x0008:0xc00018fd, DPL=0
IDT[0x06]=32-Bit Interrupt Gate target=0x0008:0xc0001916, DPL=0
IDT[0x07]=32-Bit Interrupt Gate target=0x0008:0xc000192f, DPL=0
IDT[0x08]=32-Bit Interrupt Gate target=0x0008:0xc0001948, DPL=0
IDT[0x09]=32-Bit Interrupt Gate target=0x0008:0xc0001960, DPL=0
IDT[0x0a]=32-Bit Interrupt Gate target=0x0008:0xc0001979, DPL=0
IDT[0x0b]=32-Bit Interrupt Gate target=0x0008:0xc0001991, DPL=0
IDT[0x0c]=32-Bit Interrupt Gate target=0x0008:0xc00019a9, DPL=0
IDT[0x0d]=32-Bit Interrupt Gate target=0x0008:0xc00019c2, DPL=0
IDT[0x0e]=32-Bit Interrupt Gate target=0x0008:0xc00019da, DPL=0
IDT[0x0f]=32-Bit Interrupt Gate target=0x0008:0xc00019f2, DPL=0
IDT[0x10]=32-Bit Interrupt Gate target=0x0008:0xc0001a0b, DPL=0
IDT[0x11]=32-Bit Interrupt Gate target=0x0008:0xc0001a24, DPL=0
IDT[0x12]=32-Bit Interrupt Gate target=0x0008:0xc0001a3c, DPL=0
IDT[0x13]=32-Bit Interrupt Gate target=0x0008:0xc0001a55, DPL=0
IDT[0x14]=32-Bit Interrupt Gate target=0x0008:0xc0001a6e, DPL=0
IDT[0x15]=32-Bit Interrupt Gate target=0x0008:0xc0001a87, DPL=0
IDT[0x16]=32-Bit Interrupt Gate target=0x0008:0xc0001aa0, DPL=0
IDT[0x17]=32-Bit Interrupt Gate target=0x0008:0xc0001ab9, DPL=0
IDT[0x18]=32-Bit Interrupt Gate target=0x0008:0xc0001ad2, DPL=0
IDT[0x19]=32-Bit Interrupt Gate target=0x0008:0xc0001aea, DPL=0
IDT[0x1a]=32-Bit Interrupt Gate target=0x0008:0xc0001b03, DPL=0
IDT[0x1b]=32-Bit Interrupt Gate target=0x0008:0xc0001b1b, DPL=0
IDT[0x1c]=32-Bit Interrupt Gate target=0x0008:0xc0001b33, DPL=0
IDT[0x1d]=32-Bit Interrupt Gate target=0x0008:0xc0001b4c, DPL=0
IDT[0x1e]=32-Bit Interrupt Gate target=0x0008:0xc0001b64, DPL=0
IDT[0x1f]=32-Bit Interrupt Gate target=0x0008:0xc0001b7c, DPL=0
IDT[0x20]=32-Bit Interrupt Gate target=0x0008:0xc0001b95, DPL=0
```

▲图 7-27　中断描述符 IDT 中信息

图 7-27 中第一列是中断门描述符的序号，这里共 0x20 个。在白色框之内的 target 是门描述符中所指向的中断处理程序地址，用选择子：偏移量的形式给出，所有中断处理程序所在段的选择子都是 0x0008，段内偏移地址各不相同，也就是中断门描述符中记录的目标程序的选择子及选择子所在段的偏移量。

如果是想查看某个中断门描述符，可以用“info idt 序号”的形式来查看，图 7-28 所示就是中断向量号 0x20 对应的中断门描述符。

```
<bochs:11> info idt 0x20
Interrupt Descriptor Table (base=0xc0002cc0, limit=263):
IDT[0x20]=32-Bit Interrupt Gate target=0x0008:0xc0001b95, DPL=0
```

▲图 7-28　中断向量 0x20 对应的中断门描述符

好啦，本节就这样，下一节我们让中断处理程序变得帅一点，另外，本来想给大伙儿演示进入中断时处理器实际压栈情况，思考了一下，不如在下一节一起演示比较合适，在下节中，我们实际查看一下处理器进出中断时栈中的情况。

7.6.2　改进中断处理程序

在上一节中，我们只是为了向大家演示中断机制的原理，所以中断处理程序很简陋，确切地说，我们只是在中断处理程序的入口中“处理”了一下就返回啦，一切都只是在汇编中完成的。

按理说最终干活的中断处理函数相对功能复杂，光靠汇编语言来写太不人道了，而且我们也不想把 kernel.S 写得太冗长，这样维护起来也不方便。所以最好的选择是用 C 语言编写中断处理程序，在汇编中调用它。

好啦，方案就这样定啦：在汇编版本的 intrXXentry 中调用 C 语言版本的中断处理函数。这样汇编中的 intrXXentry 就如其名一样，真正变成了中断的 entry，即入口。

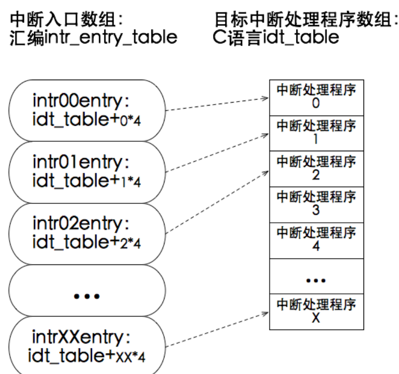
接下来的工作要自己协调好才行：在 C 语言中建立个目标中断处理函数数组 idt_table，数组元素是 C 版本的中断处理函数地址，供汇编语言中的 intrXXentry 调用。

intrXXentry 是如何找到对应的 C 版本中断处理函数呢？

由于每个中断的中断向量号不同，所以它就成了中断入口程序 intrXXentry 调用 C 版本中断处理函数的依据，idt_table 中数组元素是 32 位地址，故要占用 4 字节，这样，在中断入口程序中，将中断向量号乘以 4，再加上 idt_table 地址的值，便是对应的 C 语言版本的中断处理函数地址。

这样一来，intr_entry_table 数组元素，也就是每个中断入口 intrXXentry，都相当于用自己的中断向量号作为 idt_table 中的索引，于是 intr_entry_table 数组中的每个元素均与 idt_table 中的每个元素对等，相当于 intr_entry_table[i]调用 idt_table[i]。

这么说吧，汇编版本的中断入口程序 intrXXentry 相当于路由器，中断到达时，它根据自己所属的中断向量号把中断路由到对应的 C 版本中断处理程序。关系如图 7-29 所示。



▲图 7-29　intr_table 与 idt_table

这涉及到修改两个文件，interrupt.c 和 kernel.S，我们现在先看一下 interrupt.c 的内容，见代码 7-9。

代码 7-9　（project/c7/b/ kernel/interrupt.c）

```
…略
27 char* intr_name[IDT_DESC_CNT];          // 用于保存异常的名字
28 intr_handler idt_table[IDT_DESC_CNT];
//定义中断处理程序数组，在 kernel.S 中定义的 intrXXentry
//只是中断处理程序的入口，最终调用的是 ide_table 中的处理程序
29 extern intr_handler intr_entry_table[IDT_DESC_CNT];
//声明引用定义在 kernel.S 中的中断处理函数入口数组
…略
71 /* 通用的中断处理函数，一般用在异常出现时的处理 */
 72 static void general_intr_handler(uint8_t vec_nr) {
```

```
 73     if (vec_nr == 0x27 || vec_nr == 0x2f) {
//IRQ7 和 IRQ15 会产生伪中断（spurious interrupt），无需处理
//0x2f 是从片 8259A 上的最后一个 IRQ 引脚，保留项
 74        return;
 75     }
 76     put_str("int vector : 0x");
 77     put_int(vec_nr);
 78     put_char('\n');
 79 }
 80
 81 /* 完成一般中断处理函数注册及异常名称注册 */
 82 static void exception_init(void) {
83     int i;
84     for (i = 0; i < IDT_DESC_CNT; i++) {
 85 /* idt_table 数组中的函数是在进入中断后根据中断向量号调用的
 86  * 见 kernel/kernel.S 的 call [idt_table + %1*4] */
 87       idt_table[i] = general_intr_handler;
// 默认为 general_intr_handler
 88                  // 以后会由 register_handler 来注册具体处理函数
 89       intr_name[i] = "unknown";    // 先统一赋值为 unknown
 90     }
 91     intr_name[0] = "#DE Divide Error";
 92     intr_name[1] = "#DB Debug Exception";
 93     intr_name[2] = "NMI Interrupt";
 94     intr_name[3] = "#BP Breakpoint Exception";
 95     intr_name[4] = "#OF Overflow Exception";
 96     intr_name[5] = "#BR BOUND Range Exceeded Exception";
 97     intr_name[6] = "#UD Invalid Opcode Exception";
 98     intr_name[7] = "#NM Device Not Available Exception";
 99     intr_name[8] = "#DF Double Fault Exception";
100     intr_name[9] = "Coprocessor Segment Overrun";
101     intr_name[10] = "#TS Invalid TSS Exception";
102     intr_name[11] = "#NP Segment Not Present";
103     intr_name[12] = "#SS Stack Fault Exception";
104     intr_name[13] = "#GP General Protection Exception";
105     intr_name[14] = "#PF Page-Fault Exception";
106     // intr_name[15] 第 15 项是 intel 保留项,未使用
107     intr_name[16] = "#MF x87 FPU Floating-Point Error";
108     intr_name[17] = "#AC Alignment Check Exception";
109     intr_name[18] = "#MC Machine-Check Exception";
110     intr_name[19] = "#XF SIMD Floating-Point Exception";
111 }
112
113 /*完成有关中断的所有初始化工作*/
114 void idt_init() {
115     put_str("idt_init start\n");
116     idt_desc_init();        // 初始化中断描述符表
117     exception_init();       // 异常名初始化并注册通常的中断处理函数
118     pic_init();             // 初始化 8259A
…略
```

代码 7-9 中文件开头定义了中断异常名数组 intr_name，它的数组长度是 IDT_DESC_CNT，用来记录每一项异常的名字，这是将来在调试时用的，主要是方便咱们自己，后面还会有相关叙述。

另外还定义了中断处理函数数组 idt_table，元素个数为 IDT_DESC_CNT，这与咱们要处理的中断数量对应。此数组中的元素即目标中断处理函数地址先由函数 exception_init 初始化（之所以是“先”，这是因为以后会在初始化后，再由专门的注册函数来修改），其中的数组元素将由 intrXXentry 来调用，如图 7-29 所示。

exception_init 在第 117 行被调用，在第 82 行定义，我们先看 exception_init 的实现。

初始化是由第 84～90 行的 for 循环完成的，此循环遍历 IDT_DESC_CNT 个中断，将中断处理函数数组 idt_table 中的所有元素初始化，先都指向 general_intr_handler 函数。这个函数定义在第 72 行，用来做未处理的中断或通用的中断处理程序，意指默认的，将来咱们设置新的硬件时再通过注册函数更新它，这是后话。一会儿再说 general_intr_handler，回来继续看 exception_init。

在第 89 行，我们初始化了 intr_name 数组中的每个元素，此数组用来记录异常的名字，因为在编写操作系统的过程中，由于粗心或能力不足等原因，经常会导致处理器抛异常，咱们在这里将相应的异常名定义好，将来在异常出现时，可以根据中断向量号在 intr_name 数组中检索到相应的异常名，这样就能帮助

咱们排查错误了。你懂的，在无操作系统支持下写程序，排错有时候并不是那么顺利。由于 intr_name 是用来记录 IDT_DESC_CNT（33）个的名称，但异常只有 20 个，所以先一律赋值为“unknown”，这样就保证 intr_name[20～32]不指空了。接下来在循环体外单独为 0～19 这 20 个异常赋予正确的异常名称。正如您看到的，这就是第 91～110 行所做的事。

现在看看 general_intr_handler 函数，此函数是通用的中断处理函数，也就是相关中断若没有定义具体的中断处理函数时将调用 general_intr_handler。general_intr_handler 只接受一个参数：就是中断向量号。

在第 73～75 行，是为了单独处理伪中断（spurious interrupt）。伪中断顾名思义，并不是真正的中断，属于某种不希望发生的硬件中断。产生伪中断的原因很多，如中断线路上电气信号异常，或是中断请求设备本身有问题。

在咱们的实际情况中，它经常由 IRQ7 和 IRQ15 产生，IRQ7 是并口 1，IRQ15 是保留的。由于它们无法通过 IMR 寄存器屏蔽，所以在这里单独处理它们。这里的处理规则是直接通过 return 返回，无视它。

第 76～78 行是打印中断向量号，正常情况下将会打印“int vector：向量号换行”。当然这也只是临时的，完全是为了演示中断处理程序正常运行。一会儿我们将看到，时钟中断将会触发 0x20 的中断向量，我们通过此函数去验证。

interrupt.c 修改就到这了，接下来再看 kernel.S 有哪些改进的地方，见代码 7-10。

代码 7-10　（project/c7/b/kernel/kernel.S）

```
…略
5 extern idt_table              ;idt_table 是 C 中注册的中断处理程序数组
 6
 7 section .data
 8 global intr_entry_table
 9 intr_entry_table:
10
11 %macro VECTOR 2
12 section .text
13 intr%1entry:                  ; 每个中断处理程序都要压入中断向量号
                                ; 所以一个中断类型一个中断处理程序
                                ; 自己知道自己的中断向量号是多少
14
15    %2                        ; 中断若有错误码会压在 eip 后面
16 ; 以下是保存上下文环境
17    push ds
18    push es
19    push fs
20    push gs
21    pushad                    ; PUSHAD 指令压入 32 位寄存器，其入栈顺序是:
; EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI,EAX 最先入栈

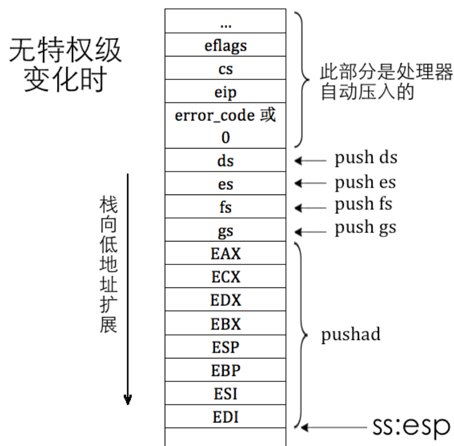
22
23    ;如果是从片上进入的中断
      ;除了往从片上发送 EOI 外，还要往主片上发送 EOI
24    mov al,0x20               ; 中断结束命令 EOI
25    out 0xa0,al               ; 向从片发送
26    out 0x20,al               ; 向主片发送
27
28    push %1                   ;不管 idt_table 中的目标程序是否需要参数
                                ;都一律压入中断向量号，调试时很方便
29    call [idt_table + %1*4]   ; 调用 idt_table 中的 C 版本中断处理函数
30    jmp intr_exit
31
32 section .data
33    dd   intr%1entry          ; 存储各个中断入口程序的地址
                                ; 形成 intr_entry_table 数组
34 %endmacro
35
36 section .text
37 global intr_exit
38 intr_exit:
39 ; 以下是恢复上下文环境
40    add esp, 4                ; 跳过中断号
41    popad
42    pop gs
43    pop fs
```

```
 44    pop es
 45    pop ds
 46    add esp, 4              ; 跳过 error_code
 47    iretd
 48
 49 VECTOR 0x00,ZERO
…略
81 VECTOR 0x20,ZERO
```

我们为了在汇编文件 kernel.S 中调用 interrupt.c 中定义的 idt_table，需要声明 idt_table，所以在代码 7-10 中开头第 5 行，用 extern idt_table 声明。

本节的 kernel.S 较上一版本变化较大的是多了第 17～21 行保护进程上下文的代码。因为在此汇编文件中要调用 C 程序，一定会使当前寄存器环境破坏，所以要保存当前所使用的寄存器环境。我们只要把 4 个段寄存器 ds、es、fs、gs 和 8 个 32 位通用寄存器保护起来就够了，这已经包括了我们所使用的全部寄存器。所以在程序中先把 ds、es、fs 和 gs 这 4 个 16 位段寄存器压栈，虽然是 16 位，但在 32 位下段寄存器压栈有点特殊，入栈后要占用 4 字节（而位于其他寄存器和内存中的值若为 16 位，则入栈后只占 2 字节）。然后再通过 pushad（push all double word register）指令压入 8 个通用寄存器。这 8 个寄存器入栈顺序是 EAX->ECX->EDX->EBX->ESP->EBP->ESI-> EDI，最先入栈的是 EAX。

为了更形象一点，我们看下此时栈中数据分布的情况，如图 7-30 所示。



▲图 7-30　执行到 pushad 后的栈中情况

在第 28 行，又把%1，即中断向量号压入栈中作为 idt_table 数组中某元素所指向的中断处理程序的参数。我们知道 idt_table 数组中记录的是 C 语言下编写的中断处理程序，这些处理程序其实就是函数，函数就有可能需要参数，但大伙儿知道并不是所有的函数都需要参数。想想看，一个中断对应一个中断向量，同理一个中断向量对应一个中断处理函数，起初我们用中断向量的目的是找到中断处理函数，已经进入中断处理函数了，那函数体中还要中断向量干吗？如果您这样想，给您点一百个赞。确实没有实际用途，但在出现异常时却很有用。您还记得我们在 interrupt.c 中的通用中断处理函数 general_intr_handler 吗？它作为默认的中断处理函数注册到了 idt_table 数组中，只有当没有新的中断处理函数覆盖它时，它才会有效，所以它通常作为没人理会的相关中断的处理程序。在前 20 个异常中，除 pagefault 可以利用外，其他异常的发生基本属于编程出错的情况，此时需要排查。所以，在我们的系统中，对于大多数异常我们也不做处理，当处理器抛出异常时，我们需要知道是哪个异常引起的，这样才能排查问题，然而这必然要用到中断向量号，然后用该向量号作为 intr_name 数组的索引，这样就能找到相应异常的名字，知道发生了哪种异常，从而方便我们调试。当然，调试这块还是属于后话，目前我们先把向量号作为参数传进来再说，程序是否需要参数，用不用参数都没关系，这不影响编译和运行，毕竟参数入栈是由我们控制的，只要我们记得将向量号出栈就好啦。

在第 29 行，通过“call [idt_table + %1*4]”便调用了对应的 C 语言编写的中断处理程序。其中“[idt_table + %1*4]” 是 32 位下的基址变址寻址，由于 idt_table 中的每个元素都是 32 位地址，故占用 4 字节大小，所以将向量号乘以 4，再加上 idt_table 数组的起始地址，便得到了下标为中断向量号%1 的数组元素地址，再对该地址通过中括号[]取值，便得到了该数组元素中所指向 C 语言编写的中断处理程序，也就是之前在 exception_init 函数中注册过的 general_intr_handler。将来某些外设需要重写中断处理程序（比如时钟、键盘、硬盘），所以该中断对应的中断处理程序将不会是 general_intr_handler，这是后话，怕大家误解以后也总是用 general_intr_handler 处理一切中断才多说了两句。

由于是用 call 指令调用的中断处理程序，所以在相应的中断处理程序执行完成后，程序流程会回到第 30 行的 jmp intr_exit，这就是说，中断处理程序执行完成了，现在该恢复程序的上下文了，这里所说的上

下文就是之前进入中断入口程序 intrXXentry 时保护的那几个寄存器。

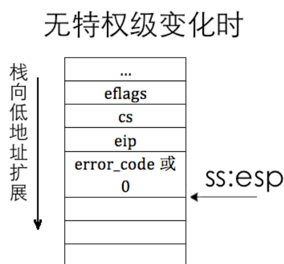
intr_exit 定义在第 38 行，其实现是先用 add esp， 4 跳过在第 28 行压入的中断处理程序的参数：中断向量号。接下来再以寄存器入栈的相反顺序依次弹出栈恢复到寄存器。

之前用 pushad 一次性压入 8 个通用寄存器，与之对应的将 8 个通用寄存器一次性出栈指令是 popad（pop all double word register），这里我要说点 popad 的注意事项。我们知道栈也是内存，对于它的读、写操作，完全可以用 mov 操作完成，但我们还得自己维护栈中最新可用位置的指针，此指针相当于 esp 的作用，但这样很麻烦。为了不再自己维护栈顶指针，处理器提供了 push 和 pop 指令，这两个指令在执行时自动维护栈顶指针 esp。pop 操作都会让 esp 的值自减一个操作数大小，popad 也是一样，无非就是使 esp 自加 8 个操作数大小，esp 会自减 8*4=32 字节，若栈中旧 esp 的值弹到 esp 寄存器就错了，所以 popad 在执行弹栈时，栈中 esp 的值会被忽略，栈中其他 7 个 32 位通用寄存器的值会被弹进相应的寄存器。

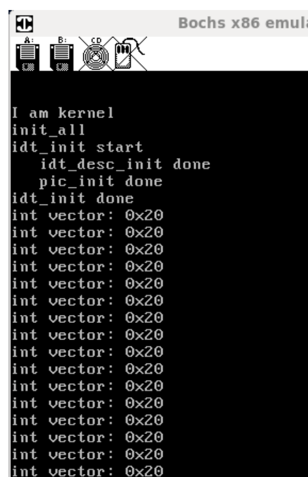
好啦 popad 的注意事项说完了，于是在 intr_exit 中先用 popad 指令将 8 个 32 位寄存器出栈，除 esp 外，其余 7 个都恢复到各自寄存器中，栈中 esp 的值弹出后被忽略。然后再将栈中 gs、fs、es、ds 的值依次出栈恢复到 gs、fs、es、ds 寄存器中。

第 46 行，此时栈指针指向栈中 error_code 的位置，当然若中断无错误码，此处是 0，如图 7-31 所示，所以需要将其跳过，这样后面的 iretd 指令执行时，栈顶指针 esp 才能指向栈中 eip，iretd 才能在栈中弹出正确的值到各寄存器。跳过的方法同跳过参数中断号一样，就是通过 add 指令把 esp 加 4。

好啦，该修改的都改完了，现在上机测试结果，编译运行还是之前那一套，结果如图 7-32 所示。



▲图 7-31　准备跳过错误码



▲图 7-32　改进后的中断处理程序

图 7-32 是运行中的结果，这里是不断打印“int vector: 0x20”。由于我们只打开了时钟中断，并且时钟的中断向量号是 0x20，效果还是符合预期的，我们成功啦。

7.6.3　调试实战：处理器进入中断时压栈出栈完整过程

本节是为了满足部分读者的求知欲和部分读者的好奇心，目的是帮助大家揭开“中断发生前处理器自动压栈、中断处理程序在栈中保护进程上下文，以及从中断返回时 iret 指令弹栈”这三个阶段所使用的栈，其中的数据是怎样变化的。此部分属于老知识的实践，不属于新知识，所以如果您对此不感兴趣，可以略过，不过您要是看了这一节，一定会了解到我的良苦用心。

我们的中断处理程序虽然成功了，对于中断发生时的处理器是如何自动压入 eflags、cs、ip 等寄存器，以及从中断返回时 iret 指令是如何将数据出栈的，这部分属于黑盒子，我想您还是感兴趣的。要不咱们趁热打铁，看看处理器进入中断时栈中的变化，顺便把 bochs 调试练习一下。

本节的目的是跟踪处理器进入中断前、中断处理过程中以及退出中断时，栈中数据是怎样的变化。所以，我们得知道处理器大概何时进入中断，这样才能找到调试的切入点。

在 bochs 中，当执行了 show int 指令时，当有中断发生时，控制台会打印出中断发生的相信信息，信息包括。

- 发生中断时执行了多少条指令，即指令数时间戳。
- 中断类型，即硬件中断，还是软中断，或是 iret 指令引起的中断。

bochs 中提供了利用指令数做断点的指令，就是 sba 或 sb，这两个断点指令是有区别的。

sb 的参数是指令数的增量，即以当前指令为准，再执行多少个指令后停下来。

sba 的参数是指令数的绝对量，也就是指令数的总量，即从处理器加电后总共执行多少条指令后停下来。

我们可以利用指令 sba 来做断点，这样便能准确地在发生中断前停下来，之后就可以再慢慢观察啦。可以利用 show int 指令来找到中断发生时的总指令数，过程如图 7-33 所示。

```
1 <bochs:1> b 0x1500
2 <bochs:2> c
3 (0) Breakpoint 1, 0xc0001500 in ?? ()
4 Next at t=18056006
5 (0) [0x000000001500] 0008:c0001500 (unk. ctxt): push ebp              ; 55
6 <bochs:3> show int
7 show interrupts tracing (extint/softint/iret): ON
8 show mask is: softint extint iret
9 <bochs:4> s 70000
10 00018056007: softint 0008:c0001501 (0xc0001501)
11 00018056007: iret 0008:c0001501 (0xc0001501)
12 Next at t=18126006
13 (0) [0x00000000151b] 0008:c000151b (unk. ctxt): jmp .-2 (0xc000151b)      ; ebfe
14 <bochs:5>
15 Next at t=18196006
16 (0) [0x00000000151b] 0008:c000151b (unk. ctxt): jmp .-2 (0xc000151b)      ; ebfe
17 <bochs:6>
18 00018240478: exception (not softint) 0008:c0001d29 (0xc0001d29)
19 00018242755: iret 0008:c000151b (0xc000151b)
20 Next at t=18266006
21 (0) [0x00000000151b] 0008:c000151b (unk. ctxt): jmp .-2 (0xc000151b)      ; ebfe
22 <bochs:7>
```

▲图 7-33　找到硬件中断发生的指令数时间戳

我们先让程序在物理地址 0x1500 处停下来，在第 1 行用 b 0x1500 便打上了断点，第 2 行的指令 c 便继续执行，直到运行到 0x1500 处停下来。

第 6 行执行 show int 指令，这样当虚拟机中有中断发生时，控制台便会打印出中断的信息。

中断是在指令执行过程中发生的，所以为了减少等待时间，咱们尽可能多执行一些指令，这里可以用单步执行指令 s 来实现，指令 s 可以后接执行的指令数，这样便一下子执行多条指令。由于我性子比较急，所以在第 9 行用单步指令 s 一下子执行了 70000 条指令。果然 show int 生效了，它在第 10 行打印了 softint 中断信息，不过这是软中断，不是外设的，咱们是为了找到时间中断，所以忽略它。

在第 14 行和第 17 行继续按回车，这将默认执行上一次的指令，即 s 70000，终于在第 18 行迎来了外部中断 exception，图中已经用下画线标出来了。我们要用的数据是冒号左边的指令数时间戳 00018240478，这表示处理器总共执行了 18240478 条指令后外设中断发生了。

当然，这是发生中断时的指令数，我们最好是在中断发生前就停下来，所以我们用 18240478-1 来做 sba 的参数，即让处理器执行了 18240477 条指令后停住。其实减多少都可以，未必减 1，减得越少，离中断发生越接近，看您的脾气快慢了。

接下来我们看看整个中断发生到退出的完整过程，如图 7-34 所示。

我们在第 1 行通过 sba 指令设置了指令数断点，这个数是中断发生时的指令数减 1。

第 3 行用 c 指令执行到断点。

由于马上要发生中断啦，处理器会马上在栈中压入 eflags、cs 和 eip 寄存器，所以在中断前先查看下栈，这样在后面就容易对比啦。在第 7 行执行 print-stack 查看栈，默认显示 16 个数据单位。第 9～24 行是栈中数据，大伙注意，在 bochs 中显示出来的栈，上面是栈顶，下面是旧数据。中间那列是栈中内存的地址，所以地址最上面最低，其是栈顶。在本例中最上面，也就是第 9 行，是栈顶，所以也就是 esp 当前值。

为了检查中断发生后压入的数据，在第 25 行用指令 r 查看寄存器。大家注意，第 35 行中显示 eflags

寄存器中的 IF 是大写的，这表示其值为 1，即开中断的状态，也就是外设中断允许发生。

```
1 <bochs:1> sba 18240477
2 Time breakpoint inserted. Delta = 18240477
3 <bochs:2> c
4 (0) Caught time breakpoint
5 Next at t=18240477
6 (0) [0x00000000151b] 0008:c000151b (unk. ctxt): jmp .-2 (0xc000151b)
7 <bochs:3> print-stack
8 Stack address size 4
9 | STACK 0xc009ffe0 [0xc0001d44]
10 | STACK 0xc009ffe4 [0x00000000]
11 | STACK 0xc009ffe8 [0x00000000]
12 | STACK 0xc009ffec [0x00000000]
13 | STACK 0xc009fff0 [0x00000000]
14 | STACK 0xc009fff4 [0x00000000]
15 | STACK 0xc009fff8 [0x00000000]
16 | STACK 0xc009fffc [0x00000000]
17 | STACK 0xc00a0000 [0xffffffff]
18 | STACK 0xc00a0004 [0xffffffff]
19 | STACK 0xc00a0008 [0xffffffff]
20 | STACK 0xc00a000c [0xffffffff]
21 | STACK 0xc00a0010 [0xffffffff]
22 | STACK 0xc00a0014 [0xffffffff]
23 | STACK 0xc00a0018 [0xffffffff]
24 | STACK 0xc00a001c [0xffffffff]
25 <bochs:4> r
26 eax: 0x20a00000 547356672
27 ecx: 0x00000000 0
28 edx: 0x0000c000 49152
29 ebx: 0x00070094 458900
30 esp: 0xc009ffe0 -1073086496
31 ebp: 0xc009fffc -1073086468
32 esi: 0x00070000 458752
33 edi: 0x00000000 0
34 eip: 0xc000151b
35 eflags 0x00000282: id vip vif ac vm rf nt IOPL=0 of df IF tf SF zf af pf cf
36 <bochs:5> s
37 Next at t=18240478
38 (0) [0x000000001d29] 0008:c0001d29 (unk. ctxt): push ds                  ; 1e
```

▲图 7-34　中断调试 a

再通过一步中断就发生啦，在第 36 行执行指令 s 后，果然下一条指令是 push ds，如图 7-35 中第 38 行的显示结果。这说明此时已经发生了中断。

```
39 <bochs:6> print-stack
40 Stack address size 4
41 | STACK 0xc009ffd0 [0x00000000]
42 | STACK 0xc009ffd4 [0xc000151b]
43 | STACK 0xc009ffd8 [0x00000008]
44 | STACK 0xc009ffdc [0x00000282]
45 | STACK 0xc009ffe0 [0xc0001d44]
46 | STACK 0xc009ffe4 [0x00000000]
47 | STACK 0xc009ffe8 [0x00000000]
48 | STACK 0xc009ffec [0x00000000]
49 | STACK 0xc009fff0 [0x00000000]
50 | STACK 0xc009fff4 [0x00000000]
51 | STACK 0xc009fff8 [0x00000000]
52 | STACK 0xc009fffc [0x00000000]
53 | STACK 0xc00a0000 [0xffffffff]
54 | STACK 0xc00a0004 [0xffffffff]
55 | STACK 0xc00a0008 [0xffffffff]
56 | STACK 0xc00a000c [0xffffffff]
57 <bochs:7> s
58 Next at t=18240479
59 (0) [0x000000001d2a] 0008:c0001d2a (unk. ctxt): push es                  ; 06
60 <bochs:8>
61 Next at t=18240480
62 (0) [0x000000001d2b] 0008:c0001d2b (unk. ctxt): push fs                  ; 0fa0
63 <bochs:9>
64 Next at t=18240481
65 (0) [0x000000001d2d] 0008:c0001d2d (unk. ctxt): push gs                  ; 0fa8
66 <bochs:10>
67 Next at t=18240482
68 (0) [0x000000001d2f] 0008:c0001d2f (unk. ctxt): pushad                   ; 60
```

▲图 7-35　中断调试 b

我们在第 39 行查看当前栈，果然处理器已经压入了寄存器的值，其中第 44 行是 eflags 的值，第 43

行是 cs 中的选择子 0x8，第 42 行是旧 eip 的值 0xc000151b，第 41 行的 0x00000000 是中断处理入口程序中第一条指令“push 0”压入的。您可以和图 7-34 中的栈信息对比下。

在第 57 行继续用 s 指令单步执行，回车后是将图 7-34 中最下面的 push ds 执行完成，接着又依次执行了 59 行的 push es、62 行的 push fs 和 65 行的 push gs。到第 66 行时，下一条要执行的指令是第 68 行的 pushad，在执行它之前，我们先查看下栈中情况。

```
69 <bochs:11> print-stack
70 Stack address size 4
71 | STACK 0xc009ffc0 [0xc0000018]
72 | STACK 0xc009ffc4 [0x00000000]
73 | STACK 0xc009ffc8 [0x00070010]
74 | STACK 0xc009ffcc [0xc0000010]
75 | STACK 0xc009ffd0 [0x00000000]
76 | STACK 0xc009ffd4 [0xc000151b]
77 | STACK 0xc009ffd8 [0x00000008]
78 | STACK 0xc009ffdc [0x00000282]
79 | STACK 0xc009ffe0 [0xc0001d44]
80 | STACK 0xc009ffe4 [0x00000000]
81 | STACK 0xc009ffe8 [0x00000000]
82 | STACK 0xc009ffec [0x00000000]
83 | STACK 0xc009fff0 [0x00000000]
84 | STACK 0xc009fff4 [0x00000000]
85 | STACK 0xc009fff8 [0x00000000]
86 | STACK 0xc009fffc [0x00000000]
87 <bochs:12> s
88 Next at t=18240483
89 (0) [0x000000001d30] 0008:c0001d30 (unk. ctxt): mov al, 0x20            ; b020
```

▲图 7-36　中断调试 c

在图 7-36 中第 69 行，通过 print-stack 查看当前栈，此时栈顶已经是 0xc009ffc0 啦，就是第 71 行所示的中间那列，较图 7-35 中的栈多了 4 个数据。第 74 行是段寄存器 ds 的值，其中低 16 位有效，即旧 ds 中选择子的值为 0x10，高 16 位是内存中的垃圾数据，第 73 行是旧 es 的选择子，它同旧 ds 的值是一样的。第 72 行是旧 fs 中选择子的值，我们没使用它，未对其初始化，所以其值为 0。第 71 行是旧 gs 中选择子的值。

在第 87 行，执行了 s 指令，所以此时执行的是上一张图中最后一行的 pushad 指令。此时再查看一下栈是否压入了 8 个通用寄存器。

为了让大家看全此次中断压入的全部数据，我们在图 7-37 的第 90 行通过 print-stack 20 打印了 20 个栈中数据。对比图 7-36 中的栈，旧栈顶是 0xc009ffc0。现在的栈顶是 0xc009ffa0，增加了从第 99～92 行之间的 8 个数据，这就是 8 个 32 位通用寄存器的值。按照压栈顺序，从第 99～92 行依次是 eax，ecx，edx，ebx，esp，ebp，esi，edi。它们的值到底对不对呢？由于从进入中断后到现在为止，我们未改变寄存器的值，只是一直在做 push 操作，所以大伙儿可以参考图 7-34 中通过 r 命令显示出的寄存器结果，它们与图 7-37 第 99～92 行中的值是一样的。顺便说一句，在 bochs 中指令 r 中显示的通用寄存器，它们从上到下的顺序，正是 pushad 指令压入它们的顺序。

```
90 <bochs:13> print-stack 20
91 Stack address size 4
92 | STACK 0xc009ffa0 [0x00000000]
93 | STACK 0xc009ffa4 [0x00070000]
94 | STACK 0xc009ffa8 [0xc009fffc]
95 | STACK 0xc009ffac [0xc009ffc0]
96 | STACK 0xc009ffb0 [0x00070094]
97 | STACK 0xc009ffb4 [0x0000c000]
98 | STACK 0xc009ffb8 [0x00000000]
99 | STACK 0xc009ffbc [0x20a00000]
100 | STACK 0xc009ffc0 [0xc0000018]
101 | STACK 0xc009ffc4 [0x00000000]
102 | STACK 0xc009ffc8 [0x00070010]
103 | STACK 0xc009ffcc [0xc0000010]
104 | STACK 0xc009ffd0 [0x00000000]
105 | STACK 0xc009ffd4 [0xc000151b]
106 | STACK 0xc009ffd8 [0x00000008]
107 | STACK 0xc009ffdc [0x00000282]
108 | STACK 0xc009ffe0 [0xc0001d44]
109 | STACK 0xc009ffe4 [0x00000000]
110 | STACK 0xc009ffe8 [0x00000000]
111 | STACK 0xc009ffec [0x00000000]
```

▲图 7-37　中断调试 d

如图 7-38 所示，继续向下执行，这次改用了指令 n，因为一会儿要调用函数，不想单步跟进去。

在图 7-38 中第 112 行的指令 n 是执行的图 7-36 中第 89 行的 mov al，0x20。这是在准备 OCW2，目的是发送 EOI。

在第 114 和第 117 分别是向主片和从片发送 OCW2，也就是发送 EOI 信号。

第 120 行的 push 0x00000020 是下一条待执行的指令，表示压入中断向量号 0x20，其对应 project/c7/b/kernel/kernel.S 的实际代码是 push %1。

第 123 行中的 call 指令对应上述 kernel.S 中的 call [idt_table + %1*4]，此时尚未执行此函数调用。

第 124 行查看下栈，此时在栈顶压入了中断向量号 0x20，如第 126 行所示。

在第 148 行显示的是即将执行的跳转指令，它是以相对跳转的形式给出的，jmp 的操作数是个偏移量，圆

括号中的是 jmp 所转移的目标绝对地址 0xc0019b0，它对应于 kernel.S 中的 jmp intr_exit。

```
112 <bochs:14> n
113 Next at t=18240484
114 (0) [0x000000001d32] 0008:c0001d32 (unk. ctxt): out 0xa0, al            ; e6a0
115 <bochs:15>
116 Next at t=18240485
117 (0) [0x000000001d34] 0008:c0001d34 (unk. ctxt): out 0x20, al            ; e620
118 <bochs:16>
119 Next at t=18240486
120 (0) [0x000000001d36] 0008:c0001d36 (unk. ctxt): push 0x00000020         ; 6a20
121 <bochs:17> n
122 Next at t=18240487
123 (0) [0x000000001d38] 0008:c0001d38 (unk. ctxt): call dword ptr ds:0xc00022e0 ; ff15e02200c0
124 <bochs:18> print-stack 20
125 Stack address size 4
126 | STACK 0xc009ff9c [0x00000020]
127 | STACK 0xc009ffa0 [0x00000000]
128 | STACK 0xc009ffa4 [0x00070000]
129 | STACK 0xc009ffa8 [0xc009fffc]
130 | STACK 0xc009ffac [0xc009ffc0]
131 | STACK 0xc009ffb0 [0x00070094]
132 | STACK 0xc009ffb4 [0x0000c000]
133 | STACK 0xc009ffb8 [0x00000000]
134 | STACK 0xc009ffbc [0x20a00000]
135 | STACK 0xc009ffc0 [0xc0000018]
136 | STACK 0xc009ffc4 [0x00000000]
137 | STACK 0xc009ffc8 [0x00070010]
138 | STACK 0xc009ffcc [0xc0000010]
139 | STACK 0xc009ffd0 [0x00000000]
140 | STACK 0xc009ffd4 [0xc000151b]
141 | STACK 0xc009ffd8 [0x00000008]
142 | STACK 0xc009ffdc [0x00000282]
143 | STACK 0xc009ffe0 [0xc0001d44]
144 | STACK 0xc009ffe4 [0x00000000]
145 | STACK 0xc009ffe8 [0x00000000]
146 <bochs:19> n
147 Next at t=18242746
148 (0) [0x000000001d3e] 0008:c0001d3e (unk. ctxt): jmp .-915 (0xc00019b0)    ; e96dfcffff
```

▲图 7-38　中断调试 e

```
149 <bochs:20>
150 Next at t=18242747
151 (0) [0x0000000019b0] 0008:c00019b0 (unk. ctxt): add esp, 0x00000004      ; 83c404
152 <bochs:21> r
153 eax: 0x00000020 32
154 ecx: 0x00000000 0
155 edx: 0x0000c000 49152
156 ebx: 0x00070094 458900
157 esp: 0xc009ff9c -1073086564
158 ebp: 0xc009fffc -1073086468
159 esi: 0x00070000 458752
160 edi: 0x00000000 0
161 eip: 0xc00019b0
162 eflags 0x00000016: id vip vif ac vm rf nt IOPL=0 of df if tf sf zf AF PF cf
163 <bochs:22> n
164 Next at t=18242748
165 (0) [0x0000000019b3] 0008:c00019b3 (unk. ctxt): popad                    ; 61
166 <bochs:23> print-stack 20
167 Stack address size 4
168 | STACK 0xc009ffa0 [0x00000000]
169 | STACK 0xc009ffa4 [0x00070000]
170 | STACK 0xc009ffa8 [0xc009fffc]
171 | STACK 0xc009ffac [0xc009ffc0]
172 | STACK 0xc009ffb0 [0x00070094]
173 | STACK 0xc009ffb4 [0x0000c000]
174 | STACK 0xc009ffb8 [0x00000000]
175 | STACK 0xc009ffbc [0x20a00000]
176 | STACK 0xc009ffc0 [0xc0000018]
177 | STACK 0xc009ffc4 [0x00000000]
178 | STACK 0xc009ffc8 [0x00070010]
179 | STACK 0xc009ffcc [0xc0000010]
180 | STACK 0xc009ffd0 [0x00000000]
181 | STACK 0xc009ffd4 [0xc000151b]
182 | STACK 0xc009ffd8 [0x00000008]
183 | STACK 0xc009ffdc [0x00000282]
184 | STACK 0xc009ffe0 [0xc0001d44]
185 | STACK 0xc009ffe4 [0x00000000]
186 | STACK 0xc009ffe8 [0x00000000]
187 | STACK 0xc009ffec [0x00000000]
```

▲图 7-39　中断调试 f

第 151 行通过 add esp，0x00000004 指令跳过了压在栈中的中断向量号（就是为调用 idt_table 中的函数所赋的参数），对应 kernel.S 中的 add esp，0x4。

第 152 行通过 r 指令查看寄存器的值，这里主要是看 esp，它目前是 0xc009ff9c。在第 163 行的指令 n

是执行的第 151 行的 add esp，0x4，此时 esp 变成了 0xc009ffa0，这一点在第 166 行的 print-stack 20 便可以看出来，第 168 行是栈顶，其值正是 0xc009ffa0。由于马上要执行第 165 行的 popad 了，所以此时才将栈打印出来给大伙儿做对比用。

```
188 <bochs:24> r
189 eax: 0x00000020 32
190 ecx: 0x00000000 0
191 edx: 0x0000c000 49152
192 ebx: 0x00070094 458900
193 esp: 0xc009ffa0 -1073086560
194 ebp: 0xc009fffc -1073086468
195 esi: 0x00070000 458752
196 edi: 0x00000000 0
197 eip: 0xc00019b3
198 eflags 0x00000096: id vip vif ac vm rf nt IOPL=0 of df if tf SF zf AF PF cf
199 <bochs:25> s
200 Next at t=18242749
201 (0) [0x0000000019b4] 0008:c00019b4 (unk. ctxt): pop gs                ; 0fa9
202 <bochs:26> r
203 eax: 0x20a00000 547356672
204 ecx: 0x00000000 0
205 edx: 0x0000c000 49152
206 ebx: 0x00070094 458900
207 esp: 0xc009ffc0 -1073086528
208 ebp: 0xc009fffc -1073086468
209 esi: 0x00070000 458752
210 edi: 0x00000000 0
211 eip: 0xc00019b4
212 eflags 0x00000096: id vip vif ac vm rf nt IOPL=0 of df if tf SF zf AF PF cf
213 <bochs:27> print-stack
214 Stack address size 4
215 | STACK 0xc009ffc0 [0xc0000018]
216 | STACK 0xc009ffc4 [0x00000000]
217 | STACK 0xc009ffc8 [0x00070010]
218 | STACK 0xc009ffcc [0xc0000010]
219 | STACK 0xc009ffd0 [0x00000000]
220 | STACK 0xc009ffd4 [0xc000151b]
221 | STACK 0xc009ffd8 [0x00000008]
222 | STACK 0xc009ffdc [0x00000282]
223 | STACK 0xc009ffe0 [0xc0001d44]
224 | STACK 0xc009ffe4 [0x00000000]
225 | STACK 0xc009ffe8 [0x00000000]
226 | STACK 0xc009ffec [0x00000000]
227 | STACK 0xc009fff0 [0x00000000]
228 | STACK 0xc009fff4 [0x00000000]
229 | STACK 0xc009fff8 [0x00000000]
230 | STACK 0xc009fffc [0x00000000]
```

▲图 7-40　中断调试 g

因为马上要执行 popad 指令啦，所以在第 188 行先用 r 指令查看下当前寄存器的值，可以在执行 popad 后做对比。

第 199 行执行了指令 s，这就是将图 7-39 中第 165 行的 popad 执行了。此时在第 202 行再次通过 r 指令查看寄存器的值，其中 eax 已经由 0x20 恢复到先前的旧值了。

在第 213 行执行了 print-stack 指令，验证下 popad 是否把栈指针增加了 8*4=0x20 字节。之前 esp 旧值是图 7-39 中的 0xc009ffa0，现在已经是 0xc009ffc0，其正好增加了 0x20。

图 7-41 中第 231 行的指令 s 是执行的图 7-40 中第 201 行的 pop gs。紧接着又依次将栈中的值 pop 到 fs、es、ds。

在第 242 行通过 add esp，4 指令跳过了栈中的 error_code 或 0，不过这是下一条待执行的指令，目前尚未执行，所以先在第 243 行查看栈中信息，栈顶在第 245 行，栈顶中的数值是 0，这正是进入中断前执行的 push 0。

图 7-42 中第 261 行的指令 s 是执行的图 7-41 第 242 行的 add esp，0x00000004，这样便跨过了栈中的 error_code 或 0，此时栈顶指向旧 eip 的值。下一个指令是第 263 行的 iretd。

在第 264 行查看栈，此时栈顶较图 7-41 中的栈顶增加了 4 字节。

由于马上要执行 iretd 指令了，这意味着 eflags、cs、eip 都要更新，所以在第 282 行打印出当前寄存器的值，用来在 iretd 执行后做对比。

在第 293 行通过 sreg 命令打印出段寄存器的值，iretd 也会改变段寄存器 cs 的值，不过由于此中断进入时未涉及到特权级转移，所以栈中旧 cs 的值和当前 cs 的值肯定是一样的。

```
231 <bochs:28> s
232 Next at t=18242750
233 (0) [0x0000000019b6] 0008:c00019b6 (unk. ctxt): pop fs                    ; 0fa1
234 <bochs:29>
235 Next at t=18242751
236 (0) [0x0000000019b8] 0008:c00019b8 (unk. ctxt): pop es                    ; 07
237 <bochs:30>
238 Next at t=18242752
239 (0) [0x0000000019b9] 0008:c00019b9 (unk. ctxt): pop ds                    ; 1f
240 <bochs:31>
241 Next at t=18242753
242 (0) [0x0000000019ba] 0008:c00019ba (unk. ctxt): add esp, 0x00000004       ; 83c404
243 <bochs:32> print-stack
244 Stack address size 4
245 | STACK 0xc009ffd0 [0x00000000]
246 | STACK 0xc009ffd4 [0xc000151b]
247 | STACK 0xc009ffd8 [0x00000008]
248 | STACK 0xc009ffdc [0x00000282]
249 | STACK 0xc009ffe0 [0xc0001d44]
250 | STACK 0xc009ffe4 [0x00000000]
251 | STACK 0xc009ffe8 [0x00000000]
252 | STACK 0xc009ffec [0x00000000]
253 | STACK 0xc009fff0 [0x00000000]
254 | STACK 0xc009fff4 [0x00000000]
255 | STACK 0xc009fff8 [0x00000000]
256 | STACK 0xc009fffc [0x00000000]
257 | STACK 0xc00a0000 [0xffffffff]
258 | STACK 0xc00a0004 [0xffffffff]
259 | STACK 0xc00a0008 [0xffffffff]
260 | STACK 0xc00a000c [0xffffffff]
```

▲图 7-41　中断调试 h

第 309 行的指令 s 执行了图 7-42 中 263 行的 iretd 指令，至此，处理器回到了被中断的进程，在第 312 行用指令 r 查看下所恢复后的寄存器值，如图 7-43 和图 7-44 所示。

```
261 <bochs:33> s
262 Next at t=18242754
263 (0) [0x0000000019bd] 0008:c00019bd (unk. ctxt): iretd                     ; cf
264 <bochs:34> print-stack
265 Stack address size 4
266 | STACK 0xc009ffd4 [0xc000151b]
267 | STACK 0xc009ffd8 [0x00000008]
268 | STACK 0xc009ffdc [0x00000282]
269 | STACK 0xc009ffe0 [0xc0001d44]
270 | STACK 0xc009ffe4 [0x00000000]
271 | STACK 0xc009ffe8 [0x00000000]
272 | STACK 0xc009ffec [0x00000000]
273 | STACK 0xc009fff0 [0x00000000]
274 | STACK 0xc009fff4 [0x00000000]
275 | STACK 0xc009fff8 [0x00000000]
276 | STACK 0xc009fffc [0x00000000]
277 | STACK 0xc00a0000 [0xffffffff]
278 | STACK 0xc00a0004 [0xffffffff]
279 | STACK 0xc00a0008 [0xffffffff]
280 | STACK 0xc00a000c [0xffffffff]
281 | STACK 0xc00a0010 [0xffffffff]
282 <bochs:35> r
283 eax: 0x20a00000 547356672
284 ecx: 0x00000000 0
285 edx: 0x0000c000 49152
286 ebx: 0x00070094 458900
287 esp: 0xc009ffd4 -1073086508
288 ebp: 0xc009fffc -1073086468
289 esi: 0x00070000 458752
290 edi: 0x00000000 0
291 eip: 0xc00019bd
292 eflags 0x00000086: id vip vif ac vm rf nt IOPL=0 of df if tf SF zf af PF cf
```

▲图 7-42　中断调试 i

由于未涉及特权转移，所以栈还是在中断处理程序中用到的那个栈，此时栈顶已经恢复为 0xc009ffe0，这和图 7-34 中进入中断前的栈顶是一样的值。

这就是中断发生时，处理器自动压栈、中断处理程序中保护上下文环境以及中断退出时 iret 指令弹栈的完整过程。

```
293 <bochs:36> sreg
294 es:0x0010, dh=0x00cf9300, dl=0x0000ffff, valid=1
295 Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed
296 cs:0x0008, dh=0x00cf9900, dl=0x0000ffff, valid=1
297 Code segment, base=0x00000000, limit=0xffffffff, Execute-Only, Non-Conforming, Accessed, 32-bit
298 ss:0x0010, dh=0x00cf9300, dl=0x0000ffff, valid=7
299 Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed
300 ds:0x0010, dh=0x00cf9300, dl=0x0000ffff, valid=1
301 Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed
302 fs:0x0000, dh=0x00001000, dl=0x00000000, valid=0
303 gs:0x0018, dh=0xc0c0930b, dl=0x80000008, valid=1
304 Data segment, base=0xc00b8000, limit=0x00008fff, Read/Write, Accessed
305 ldtr:0x0000, dh=0x00008200, dl=0x0000ffff, valid=1
306 tr:0x0000, dh=0x00008b00, dl=0x0000ffff, valid=1
307 gdtr:base=0xc0000900, limit=0x1f
308 idtr:base=0xc00020a0, limit=0x107
309 <bochs:37> s
310 Next at t=18242755
311 (0) [0x00000000151b] 0008:c000151b (unk. ctxt): jmp .-2 (0xc000151b)      ; ebfe
312 <bochs:38> r
313 eax: 0x20a00000 547356672
314 ecx: 0x00000000 0
315 edx: 0x0000c000 49152
316 ebx: 0x00070094 458900
317 esp: 0xc009ffe0 -1073086496
318 ebp: 0xc009fffc -1073086468
319 esi: 0x00070000 458752
320 edi: 0x00000000 0
321 eip: 0xc000151b
322 eflags 0x00000282: id vip vif ac vm rf nt IOPL=0 of df IF tf SF zf af pf cf
```

▲图 7-43　中断调试 j

```
323 <bochs:39> print-stack
324 Stack address size 4
325 | STACK 0xc009ffe0 [0xc0001d44]
326 | STACK 0xc009ffe4 [0x00000000]
327 | STACK 0xc009ffe8 [0x00000000]
328 | STACK 0xc009ffec [0x00000000]
329 | STACK 0xc009fff0 [0x00000000]
330 | STACK 0xc009fff4 [0x00000000]
331 | STACK 0xc009fff8 [0x00000000]
332 | STACK 0xc009fffc [0x00000000]
333 | STACK 0xc00a0000 [0xffffffff]
334 | STACK 0xc00a0004 [0xffffffff]
335 | STACK 0xc00a0008 [0xffffffff]
336 | STACK 0xc00a000c [0xffffffff]
337 | STACK 0xc00a0010 [0xffffffff]
338 | STACK 0xc00a0014 [0xffffffff]
339 | STACK 0xc00a0018 [0xffffffff]
340 | STACK 0xc00a001c [0xffffffff]
341 <bochs:40>
```

▲图 7-44　中断调试 k

7.7 可编程计数器/定时器 8253 简介

剧透一下，我们打算用计数器/定时器 8253 来设置时钟中断发生的频率，虽然单独拿一节来介绍它，但用到的并不多，所以我们并不会花太多的笔墨。另外为叙述方便，凡是用到计数器/定时器的地方，我都用定时计数器来代替。

7.7.1 时钟——给设备打拍子

尽管上一节中我们做了两个中断的例子，但我还有一件事未向大伙儿交待清楚：中断是哪来的？由谁发出的？

在揭晓答案之前，咱们先看看计算机中的时钟到底是用来干吗的。

不知道大伙儿有没有看过齐舞排练（就是一堆人跳同一种舞），在排练过程中，有个教练或领队在一直喊一、二、三、四、五、六、七、八，二、二、三、四……这是在喊拍子，目的是让大伙儿都找到“点儿”，这样队员们跟着这同一节奏跳舞，就能使整体上步调一致。如果教练没有喊这个拍子的话，由于不熟练，集体舞就乱七八糟一锅粥了。有同学会不会“抬杠”，上台表演的那些舞者也没喊拍子啊，不是也照样跳得很齐吗？其实，任何时候舞蹈都离不开拍子，舞者在表演时的拍子就是音乐，随时都在跟着音乐的节奏跳，这个音乐就是所有舞者共同的节奏，舞者们都是在用肢体表达音乐。看我说的是不是有点到位？

在计算机系统中也一样，为了使所有设备之间的通信井然有序，各通信设备间必须有统一的节奏，不能各干各的，这个节奏就称为定时或时钟。所以，大伙儿清楚了，时钟并不是计算机处理速度的衡量，而是一种使设备间相互配合而避免发生冲突的节拍。时钟只是一种时间的度量，只是一种节奏，其时间长度并不统一，各种设备都有自己的时钟，也就是说都有自己的工作节拍，比如处理器的时钟和外部设备的时钟肯定不是一个数量级，让处理器这种高速设备以外部设备低速时钟工作，处理器肯定会觉得很闲。而让低速的外部设备以处理器的时钟节拍工作，外部设备也许会急得不知所措，完全跟不上节奏。大伙儿现在应该清楚了，时钟信号并不是专指处理器的时钟，也并不特指 IRQ0 上的时钟，表达的意思仅仅是设备自己的工作节拍、频率。

计算机中的时钟，大致上可分为两大类：内部时钟和外部时钟。

内部时钟是指处理器中内部元件，如运算器、控制器的工作时序，主要用于控制、同步内部工作过程的步调。内部时钟是由晶体振荡器产生的，简称晶振，它位于主板上，其频率经过分频之后就是主板的外频，处理器和南北桥之间的通信就基于外频。Intel 处理器将此外频乘以某个倍数（也称为倍频）之后便称

为主频。处理器取指令、执行指令中所消耗的时钟周期，都是基于主频的。内部时钟是由处理器固件结构决定的，在出厂时就设定好啦，无法改变。处理器内部元件的工作速度是最快的，所以内部时钟的时间单位粒度比较精细，通常都是纳秒（ns）级的。

悄悄地说一声，内部定时是无法改变的，所以咱们对它的讨论止步于此。

外部时钟是指处理器与外部设备或外部设备之间通信时采用的一种时序，比如 IO 接口和处理器之间在 A/D 转换时的工作时序、两个串口设备之间进行数据传输时也要事先同步时钟等。外部设备的速度对于处理器来说就很慢了，所以其时钟的时间单位粒度较大，一般是毫秒（ms）级或秒（s）级的。

外部时钟和内部时钟是两套独立运行的定时体系，它们按照自己的步调协同工作。外部设备需要与处理器通信，它们之间的通信是由 IO 接口来中转完成的。问题来啦，当外部设备与处理器连接，组成了一个计算机系统后，我们就要考虑处理器与外部设备间同步数据时的时序配合问题，如何保证运行在不同时钟节拍下的设备能够同步通信。解决这个问题的大体思路是：应以处理器的内部时钟为依据来设计外部设备的时钟，既要符合处理器内部运行时序的规定，又要满足外部设备工作时序的要求。定时计数器就是用来解决时序配合问题的。大家已知道处理器的内部时钟信号由晶振产生，故计时精准稳定。但晶振产生的信号频率过高，因此必须将其送到定时计数器分频，这才能产生所需要的各种定时信号。

对于外部定时，我们有两种实现方式。一种是用软件实现，比如以下代码：

```
int cycle_cnt = 90000;
while(cycle_cnt-- > 0);
```

这个例子是让处理器执行 9 万次空循环，通过这种延迟方式达到一定的定时作用，但这种空兜处理器的代价是白白消耗时钟周期，处理器的资源是很宝贵的，可不能随意浪费。

另外一种是用硬件实现，这一类硬件称为定时器。

计时器的功能就是定时发信号。当到达了所计数的时间，计数器可以自动发一个输出信号，可以用该信号向处理器发出中断，这样处理器可以去执行相应的中断处理程序。或者用该信号直接启动某些外部设备。简单地说，其作用有点像高级开发语言中的回调函数。

和软件定时相比，硬件定时器不占用处理器，因此可以大大提升处理器利用率。

原因是硬件定时器是独立的，可以同处理器并行工作，所以用硬件定时器定时的好处是节省处理器时间。处理器给定时器设置好计数值后便可以去做别的啦，接下来的计数工作由硬件计数器独立完成。

定时器可分为不可编程定时器及可编程定时器两种，我们要接触的是第二种可编程定时器 PIC，即 Programmable Interval Timer。

常用的可编程定时计数器有 Intel 8253/8254/82C54A 等，后两个是 8253 的加强版和更强版（我自己随意说的词，反正你懂的）。由于我们只用到最基础的东西，所以咱们只介绍 8253 的用法。

8253 在名字上既称为定时器，又称为计数器，看上去像是两种功能，它到底是什么？答案是：它们是一回事。

钟表和计数器其实属于同一类物品，时间本质上就是个没有设定目标终止值的计数器，这个计数器每时每刻都在不停地计数，通常这个计数的单位是秒。所以，时间就是计数，计数也称为定时，它们本质上是一回事。

计数器是为别人提供工作的“节拍”，所以它自己必须得最可靠，这样才有资格作为时间的“基准”。为了让计数器工作时不紧不慢，有条不紊，计数器也得遵循某种“节拍”才行，这种节拍用来控制何时改变计数值（自加或自减），此节拍就是计数器的时钟脉冲信号，也称为计数脉冲信号，每一次时钟脉冲信号到来时就会修改计数值。

如何去修改计数值呢？这得看定时器的计数（计时）方式了。硬件定时器一般有两种计时的方式。

（1）正计时：每一次时钟脉冲发生时，将当前计数值加 1，直到与设定的目标终止值相等时，提示时间已到，典型的例子就是闹钟。

（2）倒计时：先设定好计数器的值，每一次时钟脉冲发生时将计数值减 1，直到为 0 时提示时间已到，典型的例子就是电风扇的定时。

我们的 8253 用的是倒计时的方式，基本上我们对它的编程就是围绕如何为其计数器赋初始的计数值。

说了半天，我们还不知道为什么要定时呢，计算机中哪些事情需要定时去做？

在微型计算机系统中定时功能是不可少的，比如为防止 RAM 中的数据丢失，每隔一段时间就要对 RAM 进行充电刷新，或者定时检测某些参数，还有最最重要的，定时向处理器发时钟中断，这就是咱们马上要做的，通过设置 8253 来调整时钟中断发生的频率。

好啦，更多精彩请见下一节。

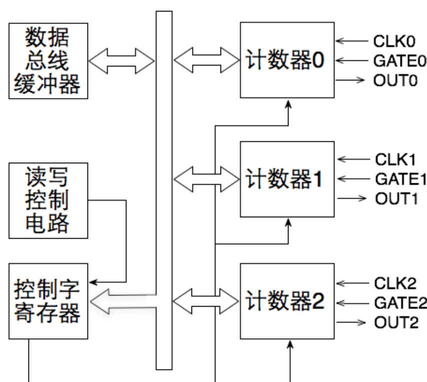
7.7.2　8253 入门

前面唠叨的已经够多了，咱们现在正式介绍下可编程定时计数器 8253，其内部逻辑结构如图 7-45 所示。

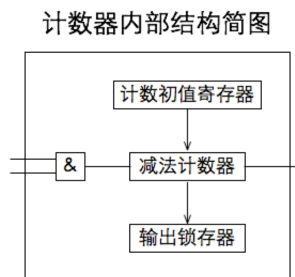
如果要把 8253 完全讲清楚，肯定要涉及到微机接口、电平信号方面的知识，对于 8253 的介绍咱们也是围绕如何对其编程来展开，如果大伙儿有深入学习的兴致，可以参考微机接口方面的资料。图 7-45 所示只是逻辑结构简图，咱们对照着它，用到什么就介绍什么。

8253 既然是计数器，那么它的价值就体现在计数方面。

在 8253 内部有 3 个独立的计数器，您看到了，就是在图 7-45 的右侧的 3 个计数器，分别是计数器 0～计数器 2，它们的端口分别是 0x40～0x42。计数器又称为通道，每个计数器都完全相同，都是 16 位大小。既然它们是独立的，也就是这三个计数器的工作是不依赖的，可以同时工作，各干各的。原因是各个计数器都有自己的一套寄存器资源，工作时自己用自己的，互不干涉。寄存器资源包括一个 16 位的计数初值寄存器、一个计数器执行部件和一个输出锁存器。其中，计数器执行部件是计数器中真正进行计数工作的元器件，其本质是个减法计数器，如图 7-46 所示。



▲图 7-45　8253 内部结构简图



▲图 7-46　计数器内部结构

在继续之前，咱们先大体看下计数器的工作原理，这样有助于大伙自上而下地理解。

每个计数器都有三个引脚：CLK，GATE，OUT。

（1）CLK 表示时钟输入信号，即计数器自己工作的节拍，也就是计数器自己的时钟频率。每当此引脚收到一个时钟信号，减法计数器就将计数值减 1。连接到此引脚的脉冲频率最高为 10MHz，8253 为 2MHz。

（2）GATE 表示门控输入信号，在某些工作方式下用于控制计数器是否可以开始计数，在不同工作方式下 GATE 的作用不同，到时候同工作方式一同介绍。

（3）OUT 表示计数器输出信号。当定时工作结束，也就是计数值为 0 时，根据计数器的工作方式，会在 OUT 引脚上输出相应的信号。此信号用来通知处理器或某个设备：定时完成。这样处理器或外部设备便可以执行相应的行为动作。

计数开始之前的计数初值保存在计数初值寄存器中，计数器执行部件（减法计数器）将此初值载入后，计数器的 CLK 引脚每收到一个脉冲信号，计数器执行部件（减法计数器）便将计数值减 1，同时将当前计数值保存在输出锁存器中。当计数值减到 0 时，表示定时工作结束，此时将通过 OUT 引脚发出信号，此信号可以用来向处理器发出中断请求，也可以直接启动某个设备工作。

下面简要说下计数器内部各个结构的功用。

计数初值寄存器用来保存计数器的初始值，它是 16 位宽度，我们对 8253 初始化时写入的计数初始值就保存在计数初值寄存器。它的作用是为计数器执行部件准备初始计数值，之后的计数过程与它无关。当计数器选择了某种重复计数的工作方式后，比如工作方式 2 和工作方式 3（后面介绍），还需要将此计数初值重新装载到计数器执行部件中。

计数器执行部件是计数器中真正“计数”的部件，计数的工作是由计数器执行部件完成的，所以它才是真正实际的计数器。8283 是个倒计时计数器，原因是计数器执行部件是个 16 位的减法计数器，它从初值寄存器中拿到起始值，载入到自己的寄存器后便开始递减计数。注意，计数过程中不断变化的值称为当前计数值，它保存在执行部件自己的寄存器中，初值寄存器中的值不受影响。

输出锁存器也称为当前计数值锁存器，用于把当前减法计数器中的计数值保存下来，其目的就是为了让外界可以随时获取当前计数值。计数器中的计数值是不断变化的，处理器无法直接从计数器中获取当前计数值。原因是这样的：计数器的使命就是通过计数的方式实现定时功能，必须要求精准，所以绝不能为了获取计数值而计数器停止计数。为了获取任意时刻的计数值，8253 只有将它送到输出锁存器，此锁存器起到暂存寄存器的作用。这样处理器便能够从输出锁存器中获取瞬间计数值。

计数初值寄存器、计数器执行部件和输出锁存器都是 16 位宽度的寄存器，所以高 8 位和低 8 位都可以单独访问。我们之后为其赋予初始计数值时就分为高 8 位和低 8 位分别操作。

三个计数器都有自己的用途，它们的作用见表 7-4。

表 7-4 8253 计数器

计数器名称	端口	作　　用
计数器 0	0x40	在个人计算机中，计数器 0 专用于产生实时时钟信号。它采用工作方式 3，往此计数器写入 0 时则为最大计数值 65536
计数器 1	0x41	在个人计算机中，计数器 1 专用于 DRAM 的定时刷新控制。PC/XT 规定在 2ms 内进行 128 次的刷新，PC/AT 规定在 4ms 内进行 256 次的刷新
计数器 2	0x42	在个人计算机中，计数器 2 专用于内部扬声器发出不同音调的声音，原理是给扬声器输送不同频率的方波

表 7-4 中简单介绍了 3 个计数器的作用，后两个咱们不用再深究了，毕竟目前用不着它们。计数器 0 可是咱们的重点，给大伙儿介绍 8253 其实就是为了讲计数器 0，这里咱们多说两句。

计数器 0 的作用是产生时钟信号，这个时钟是指连接到主片 IRQ0 引脚上的那个时钟，也就是说计数器 0 决定时钟中断信号的发生频率。

我们已经了解，计数器的工作就是以计数的形式实现定时功能，计数时间到达后就会发一个输出信号，此信号可以向处理器发中断。因此，计数器 0 的计时到期后就会发出时钟中断信号，中断代理 8259A 就会感知引脚 IRQ0 有中断信号到来。

7.7.3 8253 控制字

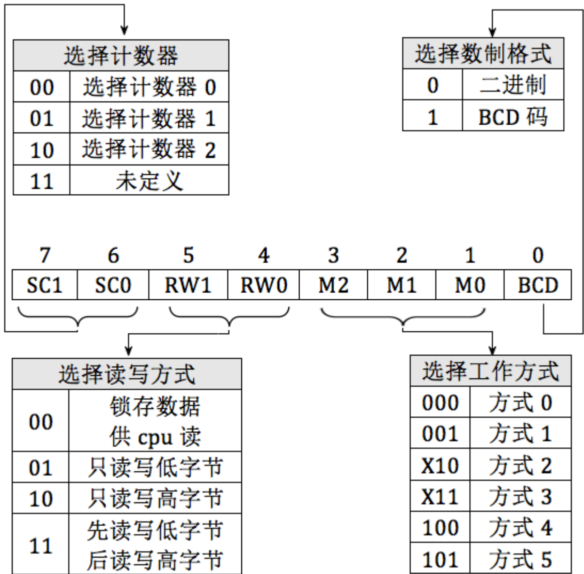
在图 7-45 中左下角是控制字寄存器，其操作端口是 0x43，它是 8 位大小的寄存器。控制字寄存器也称为模式控制寄存器，在控制字寄存器中保存的内容称为控制字，控制字用来设置所指定的计数器（通道）的工作方式、读写格式及数制，为方便叙述，我们把计数器的“工作方式、读写格式及数制”暂称为控制模式。

三个计数器是独立工作的，每个计数器都必须明确自己的控制模式才知道该怎样去工作（计数），它们各自的控制模式都要以控制字的形式在同一个控制字寄存器中设定，所以，控制字中有相关的字段来选定操作哪个计数器。

咱们先看看控制字的结构，如图 7-47 所示。

控制字由 8 位二进制组成，咱们从最高位开始介绍。

SC1 和 SC0 位是选择计数器位，即 Select Counter，或者叫选择通道位，即 Select Channel。在 8253 内部有 3 个独立的计数器，每个计数器都有自己的控制模式，但是这三个计数器的控制字是共用同一个控制字寄存器写入的，所以此处用 SC1 和 SC0 这两位去选择待操作的计数器，也就是此控制字用来设置哪个计数器。这有点像之前操作显卡时的寄存器那样，需要指定寄存器索引。这两位可组合 4 个寄存器名称，二进制 00b～10b 分别对应计数器 0～计数器 2，具体选择值如图 7-47 所示。



▲图 7-47　8253 控制字及部分说明

RW1 和 RW0 位是读/写/锁存操作位，即 Read/Write/Latch，用来设置待操作计数器（通道）的读写及锁存方式，有关锁存可以参看前面介绍的输出锁存器。计数器是 16 位宽度，当我们往计数器中写入计数初值时，或者读取计数器中的数值时，可以指定读写低 8 位，还是高 8 位。RW1 和 RW0 这两位组合成 4 种读写方式，具体选择值如图 7-47 所示。

M2～M0 这三位是工作方式（模式）选择位，即 Method 或 Mode。每个计数器有 6 种不同的工作方式，即方式 0～方式 5，后面专门介绍。

8253 的各个计数器都有两种计数方式：二进制方式和十进制方式，其中十进制方式就是用 BCD 码来表示。BCD，即 Binary-Coded Decimal，称为“二进制码的十进制数”。十进制最大数为 9，也就是需要用 4 位二进制数来表示 1 位十进制数。

BCD 位是数制位，用来指示计数器的计数方式是 BCD 码，还是二进制数。

当 BCD 位为 1 时，则表示用 BCD 码来计数，如 0x1234，则表示十进制数 1234。BCD 码的初始值范围是 0～0x9999，也就是说 BCD 码所表示的十进制范围是 0～9999。0 值则表示十进制 10000。

当 BCD 位为 0 时，则表示用二进制数来计数，如 0x1234，则表示十进制 4660。二进制数的初始值范围是 0～0xFFFF，即十进制范围是 0～65535，0 值表示 65536。

控制字部分到此结束，转战下一节。

7.7.4　8253 工作方式

介绍完了控制字，咱们花点时间介绍下工作方式。8253 一共有 6 种方式，大伙儿见表 7-5。

表 7-5　　8253 工作方式

工作方式	描　　述
方式 0	计数结束中断方式（Interrupt on Terminal Count）
方式 1	硬件可重触发单稳方式（Hardware Retriggerable One-Shot）
方式 2	比率发生器（Rate Generator）
方式 3	方波发生器（Square Wave Generator）
方式 4	软件触发选通（Software Triggered Strobe）
方式 5	硬件触发选通（Hardware Triggered Strobe）

8253 提供了多种工作方式，就说明支持多种用途，每种工作方式中，计数器的计数过程、启动停止方式都有所不同。因此，咱们要根据实际用途来选择工作方式。

计数器的计数过程在何时发生呢？将计数初值写入计数器后就开始计数了吗？不完全是。开始计数的时机与工作方式相关，计数器开始计数需要两个条件。

（1）GATE 为高电平，即 GATE 为 1，这是由硬件来控制的。

（2）计数初值已写入计数器中的减法计数器，这是由软件 out 指令控制的。

当这两个条件具备后，计数器将在下一个时钟信号 CLK 的下降沿开始计数。

介绍点相关知识，不知道大伙儿是否都学过微机接口，在数字电路中，电压的高低用逻辑电平来表示。把高于 3.5 V 的电压规定为高电平，用数字 1 表示。低于 0.3 V 的电压规定为低电平，用数字 0 表示。把电平数字由 0 变成 1 的一瞬间称为上升沿，把电平数字由 1 变成 0 的一瞬间称为下降沿。

以上这两个条件，按照“哪个未完成”来划分，可分为软件启动和硬件启动。

- 软件启动

软件启动是指上面硬件负责的条件 1 已经完成，也就是 GATE 已经为 1，目前只差软件来完成条件 2，即尚未写入计数初值，只要软件负责的条件准备好，计数器就开始启动。当处理器用 out 指令往计数器写入计数初值，减法器将此初值加载后，计数器便开始计数。工作方式 0、2、3、4 都是用软件启动计数过程。

- 硬件启动

硬件启动是指上面软件负责的条件 2 已经完成，即计数初值已写入计数器。目前只差硬件来完成条件 1 了，也就是门控信号 GATE 目前还是低电平，即目前 GATE=0，只要硬件负责的条件准备好，计数器就开始启动。GATE 引脚是由外部信号来控制的，只有当 GATE 由 0 变 1 的上升沿出现时，计数器才开始启动计数。工作方式 1、5 都是用硬件启动计数过程。

以上是说明计数器是如何启动的，那计数是如何停止的呢？根据不同的工作方式，分为强制终止和自动终止。

- 强制终止

有些工作方式中，计数器是重复计数的，当计时到期（计数值为 0）后，减法计数器又会重新把计数初值寄存器中的值重新载入，继续下一轮计数，比如工作方式 2 和工作方式 3 都是采用此方式计数，此方式常见于需要周期性发信号的场合。对于采用此类循环计数工作方式的计数器，只能通过外加控制信号来将其计数过程终止，办法是破坏启动计数的条件：将 GATE 置为 0 即可。

- 自动终止

有些工作方式中，计数器是单次计数，只要定时（计数）一到期就停止，不再进行下一轮计数，所以计数过程自然就自动终止了。比如工作方式 0、1、4、5 都是单次计数，完成后自动终止。如果想在计数过程中将其终止怎么做呢？还是用那个简单粗暴可依赖的方法，将 GATE 置 0。

下面给大伙简单说下计数器的六种工作方式。

1．方式 0：计数结束中断方式（Interrupt on Terminal Count）

方式 0 也称为“计数结束输出正跳变信号”方式，其典型应用是作为事件计数器。

在方式 0 时，对 8253 任意计数器通道写入控制字，都会使该计数器通道的 OUT 变为低电平，直到计数值为 0。当 GATE 为高电平（条件 1），并且计数初值已经被写入计数器（条件 2）后，注意，此时计数器并未开始计数，大伙儿知道，计数器有自己的工作节奏，就是时钟信号 CLK。计数工作会在下一个时钟信号的下降沿开始。方式 0 下的计数工作由软件启动，故当处理器用 out 指令将计数初值写入计数器，然后到计数器开始减 1，这之间有一个时钟脉冲的延迟。之后，CLK 引脚每次收到一个脉冲信号，减法计数器就会将计数值减 1。

当计数值递减为 0 时，OUT 引脚由低电平变为高电平，这是个由低到高的正跳变信号，此信号可以接在中断代理芯片 8259A 的中断引脚 IR0 上，所以此信号可以用来向处理器发出中断，故称为计数结束“中断”方式。

方式 0 进行计数时，计数器只是单次计数，计数为 0 时，并不会再将计数初值寄存器中的值重新载入。此方式中，门控信号 GATE 用于允许或禁止计数，当 GATE=1 时允许计数，GATE=0 时则禁止计数。

2．方式 1：硬件可重触发单稳方式（Hardware Retriggerable One-Shot）

方式 1 的典型应用是作为可编程单稳态触发器，其触发信号是 GATE，这是由硬件来控制的，故此方式称为硬件可重触发单稳方式。

在方式 1 下，由处理器将计数初值写入计数器后，OUT 引脚变为高电平。不过，无论此时 GATE 是高电平，还是低电平，计数器都不会启动计数，而是等待外部门控脉冲信号 GATE 由低到高的上升沿出现，这是由硬件启动的，之后才会在下一个时钟信号 CLK 的下降沿开始启动计数，同时会将 OUT 引脚变为低电平。此后，每当 CLK 引脚收到一个时钟脉冲信号时，在其下降沿，减法计数器便开始对计数值减 1。

OUT 引脚的低电平状态一直保持到计数为 0，当计数为 0 时，OUT 引脚产生由低到高的正跳变信号。

3．方式 2：比率发生器（Rate Generator）

方式 2 是比率发生器方式，即按照比率来分频，其典型应用就是分频器，故也称为分频器方式。在我们的应用中，也将选用此方式作为计数器的工作方式，以改变时钟中断的频率。

分频器的作用是把输入频率变成符合要求的输出频率，其作用就像个变速箱，本来在一端接入的转速很快，经过里面各种齿轮相互配合，另一端输出的转速就很慢。举个生活中的例子来解释分频器原理，比如往货车上装货物，工人 A 负责把货物从仓库中搬出来放到货车前，工人 C 负责把货物装上车，工人 B 负责统计搬上车的货物总数，为了方便统计，每当工人 A 从仓库送来 2 件货物，他才通知工人 B 来往车上搬。假如工人 A 一次从仓库搬 1 件货物，一分钟搬 10 次，工人 B 对搬来的货物计数，只要一满 2 件他就通知工人 C 搬货，也就是说工人 C 每次搬 2 件，一分钟搬 5 次。此过程中，工人 A 相当于输入脉冲信号 CLK，其频率是 10 次/分钟，工人 B 相当于分频器，它将工人 C 的搬货频率降低。工人 C 相当于输出信号 OUT，其频率是 5 次/分钟。这就是分频的道理。

当处理器把控制字写入到计数器后，OUT 端变为高电平。在 GATE 为高电平的前提下，处理器将计数初值写入后，在下一个 CLK 时钟脉冲的下降沿，计数器开始启动计数，这属于软件启动。当计数值为 1 时，OUT 端由高电平变为低电平，此低电平的状态一直到计数为 0，也就是持续一个 CLK 周期。当计数为 0 时，OUT 端又变为高电平，同时，计数初值又会被载入减法计数器，重新开始下一轮计数，从此周而复始地循环计数。

此方式的特点是计数器计数到达后，自动重新载入计数初值，不需要重新写入控制字或计数初值便能连续工作。当计数初值为 N 时，每 N 个 CLK 时钟脉冲，就会在 OUT 端产生一个输出信号，这样一来，输入信号 CLK 和输出信号 OUT 的关系是 N：1，故其作用就是个分频器。综上所述，方式 2 主要用在循环分频的场合。

4．方式 3：方波发生器（Square Wave Generator）

计数器在方式 3 下工作，就相当于一个方波发生器。

当处理器把控制字写入到计数器后，OUT 端输出高电平。在 GATE 为高电平的前提下，在处理器把计数初值写入计数器后的下一个 CLK 时钟脉冲的下降沿，计数器开始计数。

如果计数初值为偶数，在每一个 CLK 时钟脉冲发生时，计数值均减 2，当计数值为 0 时，OUT 端由高电平变为低电平，并且自动重新载入计数初值，开始下一轮计数。在新的一轮计数中，当计数值再次为 0 时，OUT 端又会变成高电平，同时再次载入计数初值，又开始一轮新的计数。

如果计数初值为奇数，并且 OUT 端为高电平，则在第一个时钟脉冲的下降沿将计数减 1，这样剩下的计数值便为偶数了，所以在之后的每个时钟脉冲，计数值都被减 2。当计数值变为 0 时，OUT 端又变成低电平，同时自动从计数初值寄存器中载入计数初值开始下一轮计数。注意，在新一轮计数中，第一个时钟脉冲会将计数值减 3，这样剩下的计数值也为偶数，之后的每个时钟脉冲都会将计数值减 2。当计数值又减为 0 时，OUT 端又重新回到高电平，同时自动从计数初值寄存器中载入计数初值，开始又一轮循环计数。

方式 3 和方式 2 类似，都是软件启动，并且 OUT 端都是周期性脉冲，用在循环计数的场合。

5．方式 4：软件触发选通（Software Triggered Strobe）

当处理器把控制字写入到计数器后，OUT 端变成高电平。在 GATE 为高电平的前提下，在处理器把计数初值写入计数器后的下一个 CLK 时钟脉冲的下降沿，计数器开始计数，所以是软件启动。

当计数值为 1 时，OUT 端由高电平变为低电平，当计数值为 0，即持续一个 CLK 时钟周期后，OUT 端又回到高电平，此时计数器停止计数。此方式和方式 0 类似，都是单次计数，只有在重新写入控制字或重新写入计数初值时才会重新开启计数。

6．方式 5：硬件触发选通（Hardware Triggered Strobe）

此方式与方式 4 类似，都是一次计数，区别是计数启动的方式不同，方式 5 是硬件启动。

方式 5 中，当处理器把控制字写入到计数器后，OUT 端变成高电平。处理器把计数初值写入计数器后，计数工作要等到外部门控脉冲信号 GATE 由低到高的上升沿出现时才开启，这是由硬件启动的。

当计数值为 1 时，OUT 端由高电平变为低电平，保持一个 CLK 周期，即计数值变为 0 时，OUT 端又变为高电平，同时停止计数。

各工作模式就介绍到这啦，可能您感到有点迷糊，不过请大伙儿放心，咱们用不着了解太细致，用的时候看看表 7-6 的总结就行了。

表 7-6　8253 工作模式总结

工作方式	计数启动方式	终止计数方法	循环计数（自动重装初值）	特　点
0	写入计数初值（软件）	GATE=0	否	用来实现定时器或对外部事件计数
1	GATE 上升沿（硬件）	—	否	用来产生单稳脉冲
2	写入计数初值（软件）	GATE=0	是	用来实现对时钟脉冲 CLK 的 N 分频
3	写入计数初值（软件）	GATE=0	是	用来产生连续的方波，也用来实现对时钟脉冲 CLK 的 N 分频
4	写入计数初值（软件）	GATE=0	否	—
5	GATE 上升沿（硬件）	—	否	—
当控制字写入计数后，在方式 0 中，计数器的 OUT 端都会变成低电平，而在另外 5 个工作方式中，计数器的 OUT 端都会变成高电平				

坦白说，我们介绍 8253 的目的就是为了提升时钟中断信号的频率，而时钟中断信号是由计数器 0 负责产生的。在介绍完 6 种工作方式后，不知道大伙儿是否自己能解决这个问题：计数器 0 多久会发一个中断信号呢？

其实这是利用了分频器的原理，将高频的输入脉冲信号 CLK 转换为低频的输出信号 OUT，此信号就是时钟中断信号。这与计数器的工作方式有关，您已经知道了，并不是所有的工作方式都能让计数器周期性地发出中断信号，假设计数器 0 工作在方式 2 下，下面介绍下中断信号产生的原理。

CLK 引脚上的时钟脉冲信号是计数器的工作频率节拍，三个计数器的工作频率均是 1.19318MHz，即一秒内会有 1193180 次脉冲信号。每发生一次时钟脉冲信号，计数器就会将计数值减 1，也就是 1 秒内会将计数值减 1193180 次 1。当计数值递减为 0 时，计数器就会通过 OUT 引脚发出一个输出信号，此输出信号用于向处理器发出时钟中断信号。一秒内会发出多少个输出信号，取决于计数值变成 0 的速度，也就是取决于计数初始值是多少。默认情况下计数器 0 的初值寄存器值是 0，即表示 65536。计数值从 65536 变成 0 需要修改 65536 次，所以，一秒内发输出信号的次数为 1193180/65536，约等于 18.206，即一秒内发的输出信号次数为 18.206 次，时钟中断信号的频率为 18.206Hz。1000 毫秒/（1193180/65536）约等于 54.925，这样相当于每隔 55 毫秒就发一次中断。这也解释了前面所讲的两个中断例子中，中断处理程序输出字符为什么显得有些慢，当然大伙不做实验的话是看不到的，我当时没说而已。不过没关系，本节的目的就是重新设定中断发生的频率，让中断发生得快一些，后面有咱们大展身手的机会。

总结一下，因为：

1193180/计数器 0 的初始计数值=中断信号的频率

所以：

1193180/中断信号的频率=计数器 0 的初始计数值

很简单是吧？只是除数和商换了个位置，一会儿咱们就用到这个结论啦。

7.7.5　8253 初始化步骤

让 8253 开始工作的方法很简单，只要我们通过控制字选择用哪个计数器，指定该计数器的控制模式，再为该计数器写入计数初值就行了。下面我们说下初始化 8253 的步骤。

1．往控制字寄存器端口 0x43 中写入控制字

用控制字为指定使用的计数器设置控制模式，控制模式包括该计数器工作时采用的工作方式、读写格式及数制。

2．在所指定使用的计数器端口中写入计数初值

计数初值要写入所使用的计数器所在的端口，即若使用计数器 0，就要把计数初值往 0x40 端口写入，若使用的是计数器 1，就要把计数初值往 0x41 端口写入，依次类推。计数初值寄存器是 16 位，高 8 位和低 8 位可单独使用，所以初值是 8 位或 16 位皆可。若初值是 8 位，直接往计数器端口写入即可。若初值为 16 位，必须分两次来写入，先写低 8 位，再写高 8 位。

好啦，至此 8253 中咱们用到的部分已经介绍完了，下面要来点真格的了，你懂的。

7.8 提高时钟中断的频率，让中断来得更猛烈一些

坦白说，我们学习 8253 的目的就是为了给 IRQ0 引脚上的时钟中断信号“提速”，使其发出的中断信号频率快一些。它默认的频率是 18.206Hz，即一秒内大约发出 18 次中断信号。我们嫌它有点慢，本节我们将对 8253 编程，使时钟一秒内发 100 次中断信号，即中断信号频率为 100Hz。

在开始之前，我们先梳理下要做的工作。

- IRQ0 引脚上的时钟中断信号频率是由 8253 的计数器 0 设置的，我们要使用计数器 0。
- 时钟发出的中断信号不能只发一次，必须是周期性发出的，也就是我们要采取循环计数的工作方式，可选的工作方式为方式 2 和方式 3，这里咱们就选择方式 2，这是标准的分频方式，这正是咱们所需要的。
- 计数器发出输出信号的频率是由计数初值决定的，所以我们要为计数器 0 赋予合适的计数初值。

好，综上所述，我们的结论是：编程 8253，通过控制字指定使用计数器 0，工作方式咱们选择方式 2，即比率发生器，并且为计数器 0 赋予合适的计数初值。这个初值是多少呢？还记得前面咱们所说的公式吧？

1193180/中断信号的频率=计数器 0 的初始计数值

咱们要设置的中断信号的频率为 100Hz，直接代入公式即可，计数器 0 的初始计数值 ＝1193180/100 约等于 11932。11932 就是计数器 0 的计数初值。

好啦，这一刻等得太久了，见代码 7-11。

代码 7-11　（project/c7/c/device/timer.c）

```
 1 #include "timer.h"
 2 #include "io.h"
 3 #include "print.h"
 4
 5 #define IRQ0_FREQUENCY      100
 6 #define INPUT_FREQUENCY     1193180
 7 #define COUNTER0_VALUE      INPUT_FREQUENCY / IRQ0_FREQUENCY
 8 #define CONTRER0_PORT       0x40
 9 #define COUNTER0_NO         0
10 #define COUNTER_MODE        2
11 #define READ_WRITE_LATCH    3
12 #define PIT_CONTROL_PORT    0x43
13
14 /* 把操作的计数器 counter_no、读写锁属性 rwl、计数器模式
      counter_mode 写入模式控制寄存器并赋予初始值 counter_value */
15 static void frequency_set(uint8_t counter_port, \
16                           uint8_t counter_no, \
17                           uint8_t rwl, \
18                           uint8_t counter_mode, \
19                           uint16_t counter_value) {
20 /* 往控制字寄存器端口 0x43 中写入控制字 */
21    outb(PIT_CONTROL_PORT, \
(uint8_t)(counter_no << 6 | rwl << 4 | counter_mode << 1));
22 /* 先写入 counter_value 的低 8 位 */
23    outb(counter_port, (uint8_t)counter_value);
24 /* 再写入 counter_value 的高 8 位 */
25    outb(counter_port, (uint8_t)counter_value >> 8);
```

```
26 }
27
28 /* 初始化PIT8253 */
29 void timer_init() {
30     put_str("timer_init start\n");
31     /* 设置8253的定时周期，也就是发中断的周期 */
32     frequency_set(CONTRER0_PORT, \
                    COUNTER0_NO, \
                    READ_WRITE_LATCH,\
                    COUNTER_MODE, \
                    COUNTER0_VALUE);
33    put_str("timer_init done\n");
34 }
```

您看，前面介绍的基础还是蛮多的，这里的实际代码却非常少。也许有同学会抱怨，讲了那么多，用的却那么少，真是浪费时间……您的心情我明白，但我在这得和大伙儿说点掏心窝子的话。完成一本书其实很容易，可是把书写好并不容易。知识是有相关性的，当学习某一方面的知识时，必然要涉及到周边相关的内容，我不能把所有人都当成各方面都精通的学者，我必须得站在那些基础相对薄弱的同学的立场上思考才行，得努力让大多数人明白我在说什么。写书确实很累，但为了写好这本书，累点我也愿意，这就是我写书的良心。明知很累也要去做，此时我的心情非常符合老罗的那句话：天生骄傲。让大伙儿见笑了，咱们继续。

我们把初始化 8253 的代码写在了文件 timer.c 中，它在 device 目录下，这是新建的目录，我们今后的设备代码都会放在此目录中。

8253 的设置是在 timer_init 函数中完成的，不过，最终做初始化工作的是 frequency_set 函数，它在第 32 行被调用，其定义在第 15 行，此函数定义了五个参数。

（1）counter_port 是计数器的端口号，用来指定初值 counter_value 的目的端口号。

（2）counter_no 用来在控制字中指定所使用的计数器号码，对应于控制字中的 SC1 和 SC2 位。

（3）rwl 用来设置计数器的读/写/锁存方式，对应于控制字中的 RW1 和 RW0 位。

（4）counter_mode 用来设置计数器的工作方式，对应于控制字中的 M2～M0 位。

（5）counter_value 用来设置计数器的计数初值，由于此值是 16 位，所以我们用了 uint16_t 来定义它。

此函数的功能是把操作的计数器 counter_no、读写锁属性 rwl、计数器工作模式 counter_mode 写入模式控制寄存器并赋予计数器的计数初值为 counter_value。

frequency_set 是在第 32 行调用的，使用的实参都是一些预先定义好的宏，在 timer.c 的文件开头是这些宏的定义。

IRQ0_FREQUENCY 是我们要设置的时钟中断的频率，我们要将它设为 100Hz。

INPUT_FREQUENCY 是计数器 0 的工作脉冲信号频率，前面介绍过。

COUNTER0_VALUE 是计数器 0 的计数初值，这是由之前的公式算出来的，当然了咱们为图省事，直接用宏来计算啦，所以 COUNTER0_VALUE 的值为 INPUT_FREQUENCY / IRQ0_FREQUENCY。

CONTRER0_PORT 是计数器 0 的端口号 0x40。

COUNTER0_NO 是用在控制字中选择计数器的号码，其值为 0，代表计数器 0，它将被赋值给函数的形参 counter_no。

COUNTER_MODE 是工作模式的代码，其值为 2，即方式 2，这是我们选择的工作方式：比率发生器。

READ_WRITE_LATCH 是读写方式，其值为 3，结合图 7-47，这表示先读写低 8 位，再读写高 8 位。原因很简单，我们要写入的初值是 16 位，按照 8253 的初始化步骤，必须先写低 8 位，后写高 8 位。

PIT_CONTROL_PORT 是控制字寄存器的端口。

第 20～25 行是 frequency_set 函数中初始化 8253 的步骤，先写入控制字，再将 16 位的计数初值的低 8 位和高 8 位分别写入计数器 0 的端口。

嗯，timer.c 就说完啦，接下来再把 timer_init 函数加在文件 init.c 中就好啦，如代码 7-12 所示。

代码 7-12 （project/c7/c/kernel/init.c）

```
1 #include "init.h"
2 #include "print.h"
```

```
 3 #include "interrupt.h"
 4 #include "../device/timer.h" //用相对路径演示头文件包含
 5
 6 /*负责初始化所有模块 */
 7 void init_all() {
 8     put_str("init_all\n");
 9     idt_init();              // 初始化中断
10     timer_init();            // 初始化 PIT
11 }
```

大伙儿看到了，timer_init 函数加在了第 10 行，这样在 main.c 中调用 init_all 时就会完成 PIT8253 的初始化。

另外，我在第 4 行中包含头文件 timer.h 是用相对路径的方式，这只是为了演示用相对路径包括文件也是没问题的，并不是一定得#include "timer.h"才行。像前三行没有用相对路径的方式，是因为在编译时我们用了-I 来指定头文件目录。以后咱们不再用相对路径的形式了，仅此演示一次。

文件都准备齐了，编译链接，这里只增加了 timer.c 文件，其他文件编译及链接方式不变。编译、链接、写入 bochs 磁盘过程如下。

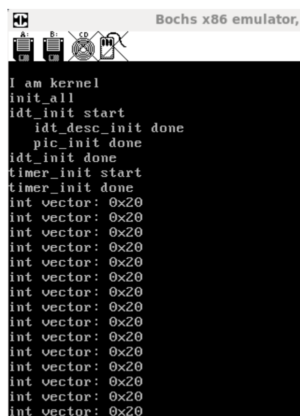
```
/* 编译 c 程序,生成目标文件 */
gcc -I lib/kernel -c -o build/timer.o device/timer.c
gcc -I lib/kernel/ -I lib/ -I kernel/ -c -fno-builtin -o build/main.o kernel/main.c
gcc -I lib/kernel/ -I lib/ -I kernel/ -c -fno-builtin -o build/init.o kernel/init.c
gcc -I lib/kernel/ -I lib/ -I kernel/ -c -fno-builtin -o build/interrupt.o \
        kernel/interrupt.c

/* 编译汇编程序,生成目标文件 */
nasm -f elf -o build/kernel.o kernel/kernel.S
nasm -f elf -o build/print.o lib/kernel/print.S

/* 链接所有目标文件,在 build 目录下生成 kernel.bin */
ld -Ttext 0xc0001500 -e main -o build/kernel.bin \
       build/main.o build/init.o build/interrupt.o \
       build/print.o build/kernel.o build/timer.o

/****将 kernel.bin 用 dd 命令写入虚拟机磁盘,
of 指定的是您机器上实际虚拟磁盘路径****/
dd if=build/kernel.bin of=/your_path/bochs/hd60M.img \
      bs=512 count=200 seek=9 conv=notrunc
```

在 bochs 中运行后，结果如图 7-48 所示。



▲图 7-48　中断频率为 100Hz

虽然图上看不出中断频率增加了，但我这明显感觉到快了，大伙儿有机会试试。

本章介绍了有关中断的内容，对于咱们来说已经够用了，整个章节到这就结束了。

第 8 章　内存管理系统

8.1 makefile 简介

随着文件越来越多，咱们要编译的文件也越来越多，每次都要手动逐个编译，然后再手动链接到一起，有没有觉得好麻烦。在上一章结尾咱们增加了 timer.c 后，要用 gcc 编译 4 个 c 文件，用 nasm 编译 2 个汇编文件，然后再通过 ld 命令将目标文件链接起来，通过 dd 命令写入，这些够麻烦的。俗话说，工欲善其事，必先利其器，没有好的编译工具咱们怎么能把精力放在编码上？考虑再三，还是花点时间给大家介绍一下 makefile，它是 Linux 下编译大型程序的工具，有了它，模块再多咱们也不会嫌麻烦。

8.1.1　makefile 是什么

通常一个大型程序是由多个程序模块文件构成的，按照其功能划分，模块文件会分布在不同的目录中。模块文件之间有包含头文件、调用函数的情况，它们之间存在依赖关系。大多数情况下，我们编写程序只是修改了某些文件，肯定不是同时更新所有文件，按理说只要把那些有过改动的文件，并且依赖于这些文件的相关文件编译即可，用不着编译全部文件。如果编译所有文件的话，对于上千万行的大型程序，通常要编译十几个小时不止。这让我想起了曾经编译全部文件的情景，下班前点一下 Visual C++的 rebuild all 按钮，第二天早上来上班的时候再看编译结果。显然，全部编译的时间成本还是蛮大的，最好还是有针对性的编译部分文件。因此，我们要知道哪些文件之间存在依赖，这样当某个文件发生变化时，我们才能找出那些受影响的文件，并且只编译它们。

既然编译全部文件过于耗时，那么问题来了，有没有办法自动针对那些有过改动的文件编译呢，这样不就高效了吗？这个问题实际上分为两个小问题。

（1）目标文件依赖哪些文件？

（2）依赖的文件是否更新？

解决了这 2 个问题，自然就可以找了相关的部分文件单独编译了。

对于第 2 个问题，倒是可以根据文件修改时间来解决，只要其修改时间比目标文件要新，就认为该文件有过更新。

对于第 1 个问题，如果光靠人工来维护文件间的依赖关系，当程序规模不大时还好，若是模块很多的时候，这些依赖关系简直会让人发狂，为了省心，我宁可编译所有文件，有时候只好这么任性。

好在万能的 Linux 提供了 make 命令，它可以帮助我们自动找出变更的文件，并根据依赖关系，找出受变更文件影响的其他相关文件，然后对这些文件按照规则进行单独处理，此处的规则一般都是指编译，如调用 gcc，但也可以是删除文件等其他行为。

上述的规则、依赖关系是定义在一个名叫 makefile 的文件中的，一看此文件名便知道它是为 make 命令准备的，makefile 文件是 make 程序的搭档，这两哥们儿的使命主要就是：发现某个文件更新后，只编译该文件和受该文件影响的相关文件，其他不受影响的文件不重新编译，从而提高了编译效率。

make 命令和 makefile 文件，它们之间是什么关系呢？

这关系类似脚本解析器和脚本语言文件，如 bin/python 和.py 文件、bin/php 及.php 文件等。makefile 相当于脚本语言文件，其中所写的内容必须遵循 make 所定义的语法，写 makefile 相当于在写脚本程序。make

程序是文件 makefile 的解析器，它定义了各种关键字、语法结构、函数、变量，甚至可以用 include 关键字包含其他 makefile。之后由 make 程序解析 makefile 中的内容，从而产生出不同的行为。

这里给大伙儿强调一下，make 和 makefile 并不是用来编译程序的，它只负责找出哪些文件有变化，并且根据依赖关系找出受影响的文件，然后执行事先在 makefile 中定义好的命令规则。只是该命令规则大多数情况是调用 gcc 或 nasm 进行编译，也有情况是执行 rm 命令删除文件，当然也可以在命令规则中执行其他命令。总之，通过 make 和 makefile 来做什么，这是由您来决定的。

说到这您也明白了，其实 make 程序只是在 makefile 中找出那些需要更新的文件，然后调用其他命令对这些文件进行处理。因为 make 就是在 shell 下执行的，所以在 makefile 中，位于命令规则里的那些命令，都是 shell 命令。

总结：依赖关系是定义在文件 makefile 中，make 程序通过解析 makefile 文件，自动找出变更的文件以及依赖此变更文件的相关文件，然后对所有受影响的相关文件执行事先定义好的命令规则。所以，咱们要做的就是把 makefile 写好。

也许有同学会说，我从来没写过 makefile，不是也照样好端端地写出了程序吗？那我打断您一下，您可能是 Windows 程序员。

由于 Windows 的易用性，对于 Windows 程序员来说，他们并不需要自己维护模块文件间的依赖关系，因为 Windows 下的集成开发环境 IDE（如 Visual C++）已经隐含地帮我们做了这些。不过对于开发者来说，这无疑是个双刃剑，一方面确实简化了操作复杂度，使开发变得容易，但另一方面却隐藏了内部细节，使得开发人员无法对程序做到完全控制，即使有的 IDE 提供了相关设置，也不能彻底控制每个细节，让 Windows 程序员常常感到力不从心啊，有木有？

虽然 Linux 下编程相对来说比 Windows 入门难一点，但人家 Linux 给咱们完全的控制力啊（甚至允许删除系统文件），这恰恰是 Linux 的魅力。人家 Linux 给用户彻底的权利，用户反而对其小心呵护，Windows 想方设法限制用户，用户却挖空心思想绕过屏障，哈哈，有点扯远了。总之，要想成为专业的程序员，就必须清楚地知道模块文件间的依赖，并且能够全程把控编译过程，makefile 正是为此而生，给咱们一个“当”程序“家”“做”程序“主”的机会。另外说一下，Linux 下还有控制力更强大的链接脚本，它的解释器是 ld 链接命令，大伙儿有兴趣自己了解一下。

8.1.2　makefile 基本语法

坦白说，makefile 的内容真心不少，多得都可以成册了，您可以通过 man make 命令看一下有多少 make 的命令行参数就能够领略一二了。

小弟在此只打算把 makefile 给大家简单介绍下，一方面的原因是我们的初衷是学习操作系统，在此过程中若总是插入一些“看似离题”的内容，真是让人很“恼火”。

另一方面，有关 makefile 的全部内容要想写好的话，怎么也得写个一两百页，何况网上有大量 makefile 的专业资料，我肯定写得不如人家详细，所以咱们依然本着“够用”的原则，用到了什么就介绍什么，我想这也是大伙儿期待的。说实在的，要想把 makefile 的知识全部学完，出于良知，我建议不要从本书中学习，咱们的内容真的是名副其实的简介，只能带您入门。makefile 毕竟只是咱们的辅助工具，所以大伙儿放心，本书一定不会用到复杂难懂的内容。

先看一下 makefile 的基本语法。

```
目标文件:依赖文件
[Tab]命令
```

makefile 基本语法包括三部分，这三部分加在一起称为一组规则，下面解释下各部分的意义。

（1）目标文件是指此规则中想要生成的文件，可以是.o 结尾的目标文件，也可以是可执行文件，也可以是个伪目标，后面会介绍伪目标。

（2）依赖文件是指要生成此规则中的目标文件，需要哪些文件。通常依赖文件不是 1 个，所以此处是个依赖文件的列表。

（3）命令是指此规则中要执行的动作，这些动作是指各种 shell 命令。命令可以有多个，但一个命令要单独占用一行，在行首必须以 Tab 开头。这是 make 规定的用法，这样 make 在解析到以 Tab 开头的行时便知道这是要执行的命令。

注意，“目标”与“依赖文件列表”之间的冒号不可少。

以上规则的意义是：要想生成目标文件，需要提前准备好依赖文件，如果依赖文件列表中任意一个文件比目标文件新，就去执行规则中的命令。

make 程序是怎样判断文件有过更新呢？先来点基础知识。

在 Linux 中，文件分为属性和数据两部分，每个文件有三种时间，分别用于记录与文件属性和文件数据相关的时间，这三个时间分别是 atime、mtime、ctime。

（1）atime，即 access time，表示访问文件数据部分时间，每次读取文件数据部分时就会更新 atime，强调下，是读取文件数据（内容）时改变 atime，比如 cat 或 less 命令查看文件就可以更新 atime，而 ls 命令则不会。

（2）ctime，即 change time，表示文件属性或数据的改变时间，每当文件的属性或数据被修改时，就会更新 ctime，也就是说 ctime 同时跟踪文件属性和文件数据变化的时间。

（3）mtime，即 modify time，表示文件数据部分的修改时间，每次文件的数据被修改时就会更新 mtime。在上面说过啦，ctime 也跟踪数据变化时间，所以，当文件数据被修改时，mtime 和 ctime 一同更新。

这三个时间可以用 stat 命令查看，如图 8-1 所示。

```
[work@localhost makefile]$ stat 1
  File: "1"
  Size: 0            Blocks: 0          IO Block: 4096   普通空文件
Device: fd00h/64768d Inode: 823447      Links: 1
Access: (0664/-rw-rw-r--)  Uid: (  501/    work)   Gid: (  501/    work)
Access: 2014-12-16 14:14:30.463236005 +0800
Modify: 2014-12-16 14:14:30.463236005 +0800
Change: 2014-12-16 14:14:30.463236005 +0800
```

▲图 8-1　atime、mtime、ctime

图中的方框中，Access 表示 atime，Modify 表示 mtime，Change 表示 ctime。

以上三个时间中最主要的就是 mtime 和 ctime，其中 mtime 只记录文件数据部分变化的时间，ctime 记录的是文件属性及数据这两部分的变化时间，其中任意部分有改变，都会更新 ctime，所以说，mtime 的更新必然伴随着 ctime 的更新。单纯考察文件数据部分改变时间的话还是 mtime 最准确。

对于文件来说，我们主要关注的是其数据部分，所以只要 make 程序分别获取依赖文件和目标文件的 mtime，对比依赖文件的 mtime 是否比目标文件的 mtime 新，就知道是否要执行规则中的命令，尽管此命令通常是编译目标文件，但大伙儿一定要清楚，make+makefile 的使命是执行规则中的命令，此处的命令是咱们来决定的，未必都是编译命令。

给大伙儿举个例子，以下是 makefile 的内容。

```
             cat -n makefile
 1     1:2
 2           echo "makefile test ok"
```

此例子中 makefile 就两行，足够简单了。第 1 行中，目标文件是文件 1，依赖文件是文件 2。第 2 行是要执行的命令。

此 makefile 的意义是如果文件 2 的 mtime 比文件 1 的 mtime 要新，就调用 echo 命令打印字符串“makefile test ok”。

为配合此例子，提前用 touch 命令生成了文件 1 和文件 2。touch 命令用来把文件的 mtime 和 atime 更新为当前时间，如果该文件不存在，则会自动生成该文件。现在用 stat 命令查看一下它们的 mtime，如图 8-2 所示。

图中标下画线的部分是 mtime，我们看到，由于文件 2 是后被 touch 的，所以文件 2 的 mtime 比文件 1 要新，按理说，此时执行 make 命令会打印字符串“makefile test ok”。

好啦，可以执行 make 啦，效果如图 8-3 所示。

除了如咱们的预期，成功打印了字符串“makefile test ok”以外，还附带了“赠品”，把 echo 这条命令本身也打印出来了。这样的效果确实不太好，咱们只想看到命令执行的结果。好在 make 给咱们提供了方法，可以在命令之前加个字符'@'，这样就不会输出命令本身信息了，这有点像 DoS 的批处理命令中“@echo off”的意思，如图 8-4 所示。

```
[work@localhost makefile]$ touch 1
[work@localhost makefile]$ touch 2
[work@localhost makefile]$ stat 1
  File: "1"
  Size: 0               Blocks: 0          IO Block: 4096   普通空文件
Device: fd00h/64768d    Inode: 823447      Links: 1
Access: (0664/-rw-rw-r--)  Uid: (  501/    work)   Gid: (  501/    work)
Access: 2014-12-16 14:14:30.463236005 +0800
Modify: 2014-12-16 14:14:30.463236005 +0800
Change: 2014-12-16 14:14:30.463236005 +0800
[work@localhost makefile]$ stat 2
  File: "2"
  Size: 0               Blocks: 0          IO Block: 4096   普通空文件
Device: fd00h/64768d    Inode: 823464      Links: 1
Access: (0664/-rw-rw-r--)  Uid: (  501/    work)   Gid: (  501/    work)
Access: 2014-12-16 14:14:33.000170845 +0800
Modify: 2014-12-16 14:14:33.000170845 +0800
Change: 2014-12-16 14:14:33.000170845 +0800
[work@localhost makefile]$
```

▲图 8-2　文件 1 和 2 的 mtime

```
[work@localhost makefile]$ make
echo "makefile test ok"
makefile test ok
[work@localhost makefile]$
```

▲图 8-3　第一个 makefile 示意

您看，这下不打印命令了。通过这个例子您也看出，make 和 makefile 确实并不是专门用来编译或生成文件的，它仅仅是执行规则中的命令。

makefile 的文件名也并非固定，可以在执行 make 时用-f 参数来指定。如果未用-f 指定，默认情况下，make 会先去找名为 GNUmakefile 的文件，若该文件不存在，再去找名为 makefile 的文件，若 makefile 也不存在，最后去找名为 Makefile 的文件。图 8-5 只是演示最高优先级的 GNUmakefile，其他两个大伙儿有兴趣的话自己试试吧。

```
[work@localhost makefile]$ cat -n makefile
     1  1:2
     2          @echo "makefile test ok"
[work@localhost makefile]$ make
makefile test ok
[work@localhost makefile]$
```

▲图 8-4　makefilek @的作用

```
[work@localhost makefile]$ ls
1  2  GNUmakefile  makefile  Makefile
[work@localhost makefile]$ cat -n GNUmakefile
     1  1:2
     2          @echo "GNUmakefile test ok"
[work@localhost makefile]$ cat -n makefile
     1  1:2
     2          @echo "makefile test ok"
[work@localhost makefile]$ cat -n Makefile
     1  1:2
     2          @echo "Makefile test ok"
[work@localhost makefile]$ make
GNUmakefile test ok
[work@localhost makefile]$
```

▲图 8-5　GNUmakefile 优先级最高

8.1.3　跳到目标处执行

makefile 中有很多目标时，我们可以用目标名称作为 make 的参数，采用“make 目标名称”的方式，单独执行目标名称处的规则。注意，这种方式只会执行目标名称处的规则，之后就退出了，后面即使存在其他的目标也不会执行，如图 8-6 所示。

```
[work@localhost new]$ cat -n makefile
     1  t1:1
     2          @echo "target1"
     3  t2:1
     4          @echo "target2"
[work@localhost new]$ make t1
target1
[work@localhost new]$ make t2
target2
[work@localhost new]$
```

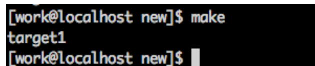
▲图 8-6　make 执行目标

makefile 中只有 2 个目标，t1 和 t2。通过“make t1”的方式解析 makefile，make 直接执行目标 t1 处的命令 echo“target1”，输出了字符串 target1，然后就退出了。接着通过“make t2”输出了字符串 target2。

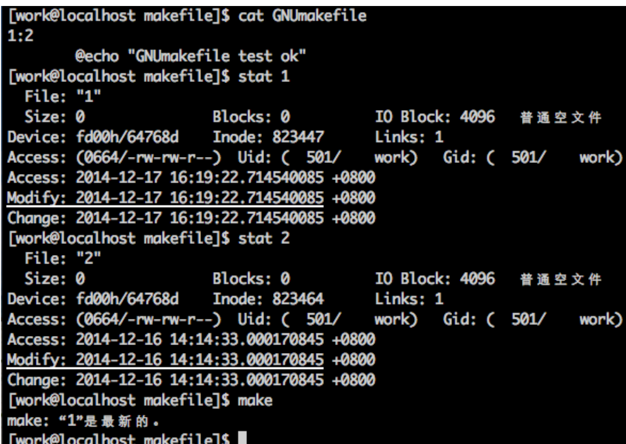
当 make 后面没有目标名称做参数时，make 会在 makefile 中第一个出现的目标处开始执行，如图 8-7 所示。

一般情况下，命令能否执行是要看所依赖文件的 mtime 是否比目标文件要新。如果依赖文件的 mtime 比目标文件旧的话，说明目标文件已经是最新的，根本不需要更新，所以按理说，此种情况下规则中的命令是不会执行的，事实也正是如此，如图 8-8 所示。

文件 1 的时间已经变成了 12-17 号，比文件 2 的时间更新，所以 make 提示：“1”是最新的，未执行 echo 命令。



▲图 8-7 make 从第 1 个目标处执行



▲图 8-8 目标文件较新，命令不执行

8.1.4 伪目标

通过上面的例子您看到了，规则中的命令并不总是被执行，有时候我们并不关心是否产生真实的目标文件，我们只希望 make 不要考虑 mtime，而是总能去执行一些命令。

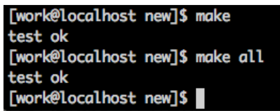
对于这个需求还是有办法的，make 规定，当规则中不存在依赖文件时，这个目标文件名就称为——伪目标。

伪目标，顾名思义，也就是不产生真实的目标文件，所以当然也就不需要依赖文件了。于是，伪目标所在的规则就变成了纯粹地执行命令，只要给 make 指定该伪目标名做参数，就能让伪目标规则中的命令直接执行。

举个例子，以下是 makefile 内容。

```
1    all:
2          @echo "test ok"
```

由于 makefile 中仅有这一个目标 all，所以如果此时执行 make all 或 make，程序只会输出 test ok，如图 8-9 所示。



▲图 8-9 伪目标演示

注意，伪目标不能和真实目标文件同名，否则就失去伪目标的意义了，为了避免伪目标和真实目标文件同名的情况，可以用关键字“.PHONY”来修饰伪目标，格式为“.PHONY:伪目标名”，这样不管与伪目标同名的文件是否存在，make 照样执行伪目标处的命令。

通常需要显式用.PHONY 修饰伪目标的场合是删除编译过程中的.o 文件，这是为了避免因旧的.o 文件已存在而影响编译。如果您在 Linux 下有过编译源码的经验，就会了解 make clean 的作用了，通常 clean 就是伪目标，用来删除编译过程中的.o 文件，如：

```
1 .PHONY:clean
2 clean:
3         rm ./build/*.o
```

伪目标的命名并没有固定的规则，用户可以按照自己的意愿定义成自己喜欢的名字。不过，由于 makefile 已经流传很广泛了，对于伪目标的命名，业界内已经有了一些约定俗成的规则，大伙儿把类似功能的伪目标定义成了同一个名字。比如上面提到的 clean，这个伪目标名称也是大伙儿公认的，它的功能通常就是清空目标文件，当然了，相应的命令部分还得是 rm 等清除相关的命令。这里再列举一些其他公认的伪目标名，见表 8-1。

表 8-1 约定俗成的伪目标名称

伪目标名称	功 能 描 述
all	通常用来完成所有模块的编译工作，类似于 rebuild all
clean	通常用于清空编译完成的所有目标文件，一般用 rm 命令实现
dist	通常用于将打包文件后的 tar 文件再压缩成 gz 文件

续表

伪目标名称	功 能 描 述
Install	通常将编译好的程序复制到安装目录下，此目录是在执行 configure 脚本通过—prefix 参数配置的
printf	通常用于打印已经发生改变的文件
tar	通常用于将文件打包成 tar 文件，也就是所谓的归档文件
test	通常用于测试 makefile 流程

8.1.5　make：递归式推导目标

在 makefile 中的目标，是以递归的方式逐层向上查找目标的，就好像是从迷宫的出口往回找来时的路一样，由果寻因，逐个向上推导。这一点尤其体现在多个目标相互依赖的情况下。举例子之前，先看两个例子文件。

```
[work@localhost new]$ cat -n test1.c
     1 void my_print(char*);
     2 void main() {
     3    my_print("hello,world\n");
     4 }
[work@localhost new]$ cat -n test2.c
     1 #include <stdio.h>
     2 void my_print(char* str) {
     3    printf(str);
     4 }
[work@localhost new]$
```

以上是 test1.c 和 test2.c 的代码，test1 中调用 test2.c 中的函数 my_print 实现打印字符，而 my_print 用系统函数 printf 实现打印。代码没法再简单了，没法再解释了。

现在想把这两个文件编译成 test.bin，那咱们为它们写个 makefile，一会儿用 make 这把“牛刀”对这两个小文件小试身手。内容如下。

```
1   test2.o:test2.c
      2      gcc -c -o test2.o test2.c
      3 test1.o:test1.c
      4      gcc -c -o test1.o test1.c
      5 test.bin:test1.o test2.o
      6      gcc -o test.bin test1.o test2.o
      7 all:test.bin
      8      @echo "compile done"
```

此 makefile 还是蛮简单的，第 1～4 行都是在准备.o 目标文件，第 5～6 行是将.o 文件生成二进制文件 test.bin。第 7 行的目标 all 是为了编译 test.bin，咱们要执行的命令是 make all，借此分析下 make 的执行流程。

（1）make 未找到文件 GNUmakefile，便继续找文件 makefile，找到后，根据命令的参数 all，从文件中找到 all 所在的规则。

（2）make 发现 all 的依赖文件 test.bin 不存在，于是就去找以 test.bin 为目标文件的规则。

（3）在第 5 行终于找到了 test.bin 的规则，但 make 发现，test.bin 的依赖文件 test1.o 和 test2.o 都不存在，于是先去找以 test1.o 为目标文件的规则。

（4）同样经过千辛万苦，在第 3 行找到了生成 test1.o 的规则，但它的依赖文件是 test1.c，由于 test1.o 本身不存在，所以不用再查看 test1.c 的 mtime，直接执行此规则的命令，即第 4 行的 gcc -c -o test1.o test1.c，用 test1.c 来编译 test1.o。

（5）生成 test1.o 后，执行流程返回到 test.bin 所在的规则，即第 5 行，此时 make 发现 test2.o 也不存在，于是继续递归查找目标 test2.o。

（6）同样，在第 1 行发现 test2.o 所在的规则，由于 test2.o 本身不存在，也不再检查其所依赖文件 test2.c 的 mtime，直接执行规则中的编译命令 gcc -c -o test2.o test2.c 生成 test2.o。

（7）生成 test2.o 后，此时执行流程又回到了第 5 行，make 发现两个依赖文件 test1.o 和 test2.o 都准备齐了，于是执行本规则的命令，即第 6 行的 gcc -o test.bin test1.o test2.o，将这两个目标文件生成可执行文件 test.bin。

（8）test.bin 终于生成了，此时回到了第 2 步目标 all 所在的规则，于是执行所在规则中的命令@echo "compile done"，打印字符串表示编译完成。提醒一下，虽然 all 被当作了真实目标文件来处理，但我们给出的命令并不是为了生成它，所以它同伪目标的作用类似，大伙儿不要感到奇怪。因为在前面我们已经解释过啦，make+makefile 并不是为了编译或生成文件，它们只是为了执行规则中的命令，无所谓命令是什么。

好啦，执行下 make all，以事实说话，效果如图 8-10 所示。

```
[work@localhost new]$ ls
makefile  test1.c  test2.c
[work@localhost new]$ make all
gcc -c -o test1.o test1.c
gcc -c -o test2.o test2.c
gcc -o test.bin test1.o test2.o
compile done
[work@localhost new]$ ls
makefile  test1.c  test1.o  test2.c  test2.o  test.bin
[work@localhost new]$ ./test.bin
hello,world
[work@localhost new]$
```

▲图 8-10　make 递归查找目标

图 8-10 中，最上面先用 ls 查看下当前目录下的文件，就这两个 test 文件和 makefile。

接着如大伙儿所愿执行了 make all，程序打印了三行 gcc 的编译命令，我故意没在 gcc 前面加字符@，这样方便大伙儿查看确实执行过编译。

大家看，先输出的是“gcc -c -o test1.o test1.c”，这与我们前面描述的相符，在生成 test.bin 的过程中，先去找生成依赖文件 test1.o 的规则。接着是输出“gcc -c -o test2.o test2.c”，完成了 test2.o 的生成，最后输出的是“gcc -o test.bin test1.o test2.o”，说明递归查找目标结束，此时具备了生成 test.bin 的条件。

生成了 test.bin 后，最后输出 compile done，编译完成。

按理说在执行 make 后，目标下会多出 test1.o、test2.o 和 test.bin 这三个文件，于是如您所愿，我又执行了一次 ls 命令，果然目录下有了这三个文件。

在图 8-10 的最后执行了 test.bin，输出 hello,world，编译执行成功。

8.1.6　自定义变量与系统变量

makefile 既然可称为编程，它必然就具备程序语言的必须的基本功能，比如，可以在 makefile 中定义变量。

变量定义的格式:变量名=值（字符串），多个值之间用空格分开。make 程序在处理时会用空格将值打散，然后遍历每一个值。另外，值仅支持字符串类型，即使是数字也被当作字符串来处理。

变量引用的格式：$(变量名)。这样，每次引用变量时，变量名就会被其值（字符串）替换。

注意，虽然变量的值会被当作字符串类型处理，但不能将其用双引号或单引号括起来，否则双引号或单引号也会被当作变量值的一部分。比如 var = 'file.c'，var 的值并不是 file1.c，而是'file.c'。当引用变量$(var) 做依赖文件时，make 会去找名为'file.c'的目标，而不是 file.c。

举个例子，在 makefile 中用变量定义目标文件名。

```
1 test2.o:test2.c
2            gcc -c -o test2.o test2.c
3 test1.o:test1.c
4            gcc -c -o test1.o test1.c
5 objfiles = test1.o test2.o
6 test.bin:$(objfiles)
7            gcc -o test.bin $(objfiles)
8 all:test.bin
9            @echo "compile done"
```

在第 5 行，定义了变量 objfiles，其值为 test1.o test2.o，此变量应用在第 6～7 行。执行 make all，效果如图 8-11 所示。

效果一样，执行成功。

除了用户自定义的变量外，make 还自行定义了一些系统级的变量，按其用途可分为命令相关的变量及参数相关的变量。这两种系统变量见表 8-2。

```
[work@localhost new]$ rm *.o test.bin
[work@localhost new]$ ls
makefile  test1.c  test2.c
[work@localhost new]$ cat -n makefile
     1  test2.o:test2.c
     2          gcc -c -o test2.o test2.c
     3  test1.o:test1.c
     4          gcc -c -o test1.o test1.c
     5  objfiles = test1.o test2.o
     6  test.bin:$(objfiles)
     7          gcc -o test.bin $(objfiles)
     8  all:test.bin
     9          @echo "compile done"
[work@localhost new]$ make all
gcc -c -o test1.o test1.c
gcc -c -o test2.o test2.c
gcc -o test.bin test1.o test2.o
compile done
[work@localhost new]$ ./test.bin
hello,world
[work@localhost new]$ 
```

▲图 8-11　make 中定义变量

表 8-2　　命令相关及参数相关的系统变量—命令相关的系统变量（部分）

变　量　名	描　　述
AR	打包程序，默认是“ar”
AS	汇编语言编译器，默认是“as”
CC	C 语言编译器，默认是“cc”
CXX	C++语言编译器，默认是“g++”
CPP	C 预处理器，默认是“$(CC) –E”，如 gcc -E
FC	Fortran 的编译器和预处理器，Ratfor 的编译器，默认是“f77”
GET	从 SCCS 文件中提取文件程序，默认是“get”
PC	Pascal 语言编译器，默认是“pc”
MAKEINFO	将 texinfo 文件转换为 info 文件，默认是“makeinfo”
RM	删除命令，默认是“rm -f”
TEX	从 TeX 源文件中创建 TexDVI 文件的程序，默认是“tex”
WEAVE	将 Web 转换为 TeX 的程序，默认是“weave”
YACC	处理 C 程序的 Yacc 词法分析器，默认是“yacc”
YACCR	处理 Ratfor 程序的 Yacc 词法分析器，默认是“yacc -r”
参数相关的系统变量（部分），几乎都无默认值	
ARFLAGS	打包程序$(AR) 的参数，默认值为 rv
ASFLAGS	汇编语言编译器参数
CFLAGS	C 语言编译器参数
CXXFLAGS	C++编译器参数
CPPFLAGS	C 预处理器参数
FFLAGS	Fortran 语言编译器参数
LDFLAGS	链接器参数
PFLAGS	Pascal 语言编译器参数
YFLAGS	Yacc 词法分析器参数

在命令相关的系统变量是有默认值的，一般参数相关的变量没有默认值。咱们在 makefile 中将它们打印出来看看是不是这样，如图 8-12 所示。

图 8-12 左边是自定义的 makefile 名称——my_makefile，里面定义了个伪目标 all，就是为了执行 echo 语句打印变量名及其值。图中右边的是执行结果，在 make 执行时要用-f 参数指定 my_makefile，这样 make 程序才不会自动去找那三个标准的文件名。您看到了，命令相关的系统变量是有值的，但参数相关的系统

变量只有 ARFLAGS 有值。

```
[work@localhost new]$ cat -n my_makefile
     1  all:
     2          @echo AR $(AR)
     3          @echo AS $(AS)
     4          @echo CC $(CC)
     5          @echo CXX $(CXX)
     6          @echo CPP $(CPP)
     7          @echo FC $(FC)
     8          @echo GET $(GET)
     9          @echo PC $(PC)
    10          @echo MAKEINFO $(MAKEINFO)
    11          @echo RM $(RM)
    12          @echo YACC $(YACC)
    13          @echo "_____ cmd var end_______"
    14          @echo ARFLAGS $(ARFLAGS)
    15          @echo ASFLAGS $(ASFLAGS)
    16          @echo CFLAGS $(CFLAGS)
    17          @echo CXXFLAGS $(CXXFLAGS)
    18          @echo CPPFLAGS $(CPPFLAGS)
    19          @echo FFLAGS $(FFLAGS)
    20          @echo LDFLAGS $(LDFLAGS)
    21          @echo PFLAGS $(PFLAGS)
    22          @echo YFLAGS $(YFLAGS)
    23          @echo "_____ para var end_______"
[work@localhost new]$
```

```
[work@localhost new]$ make -f my_makefile
AR ar
AS as
CC cc
CXX g++
CPP cc -E
FC f77
GET get
PC pc
MAKEINFO makeinfo
RM rm -f
YACC yacc
_____ cmd var end_______
ARFLAGS rv
ASFLAGS
CFLAGS
CXXFLAGS
CPPFLAGS
FFLAGS
LDFLAGS
PFLAGS
YFLAGS
_____ para var end_______
[work@localhost new]$
```

▲图 8-12　打印系统变量

这些系统变量的值并不是固定的，可以通过重新为其赋值的方式修改。

8.1.7　隐含规则

在编写规则时，若一行写不下，可以在行尾添加反斜杠字符'\'，这样下一行的内容便被认为是同一行，其实这是很多编译器和解释器都支持的功能，不仅是 make 才这样。

makefile 中另一个必须的功能是注释，如同 shell 脚本一样，makefile 中用#来单行注释，只要各行第一个非空字符（除空格、tab）是'#'，本行内容便被注释了。如果在行尾是反斜杠字符'\'，这表示下一行也应被处理为当前行，所以，连同下一行也被注释掉。

把刚才的 makefile 修改下，将前两个.o 文件的规则注释，内容如下。

```
1 #test2.o:test2.c
2 #          gcc -c -o test2.o test2.c
3 #test1.o:test1.c
4 #          gcc -c -o test1.o test1.c
5 objfiles = test1.o test2.o
6 test.bin:$(objfiles)
7           gcc -o test.bin $(objfiles)
8 all:test.bin
9           @echo "compile done"
```

先把当前目录下的*.o 和 test.bin 删除，再次执行 make all 看看有什么不同，如图 8-13 所示。

```
[work@localhost new]$ rm *.o test.bin
[work@localhost new]$ ls
makefile  my_makefile  test1.c  test2.c
[work@localhost new]$ make all
cc    -c -o test1.o test1.c
cc    -c -o test2.o test2.c
gcc -o test.bin test1.o test2.o
compile done
[work@localhost new]$ ./test.bin
hello,world
[work@localhost new]$
```

make自动用cc命令编译.c为.o文件

▲图 8-13　make 隐含规则

大伙儿看，我们原本已经在 makefile 中注释掉了前两个.o 文件的 gcc 编译命令，但在图中执行了 make all 后，make 还是输出了“cc -c -o test[12].o test[12].c”，就是图 8-13 中方框中的部分。也就是说，make 帮我们自动生成了 test1.o 和 test2.o，有没有觉得 make 还是蛮人性化的？顿时感到皇恩浩荡，哈哈。

makefile 中的编译命令是 gcc，此处输出的编译命令是 cc，所以 makefile 中的 gcc 真的是注释掉了，这一点请大伙儿放心。另外，cc 其实就是 gcc 的软链接，这两个是同一个程序，都是指向/usr/bin/gcc。

make 之所以如此宅心仁厚，是因为它有一些隐含规则。

什么是隐含规则？对于一些使用频率非常高的规则，make 把它们当成是默认的，不需要显式地写出来，当用户未在 makefile 中显式定义规则时，将默认使用隐含规则进行推导。

隐含规则对于不同的程序语言是不同的，是根据一般的依赖关系来自动推导，属于重建目标文件的通

用方法。

隐含规则只限于那些编译过程中基本固定的依赖关系，比如 C 语言代码文件扩展名为.c，编译生成的目标文件扩展名是.o，这一般是一对一的。而一个可执行程序可能是由多个.o 文件共同链接生成的，所以，从可执行程序到.o 文件的关系有可能是一对多，这种不确定性无法使之成为隐含的规则。所以，对于 C 语言的依赖关系是：文件名.o 依赖于文件名.c，仅限于源文件生成.o 目标文件，不存在.o 文件生成可执行程序的隐含规则。

总的说来，针对不同的编程语言依赖关系，make 程序通过除扩展名之外的文件名部分，再根据隐含规则，可以推导出最终的可执行文件。也就是说，若想通过隐含规则自动推导生成目标，存在于文件系统上的文件，除扩展名之外的文件名部分必须相同。比如 x.o 的 C 源文件必须名为 x.c，这样通过隐含规则才能成功生成 x.o。

隐含规则是用系统变量来实现的，比如咱们例子中用到了命令变量 CC 及参数变量 CFLAGS 及 CPPFLAGS，变量 CC 的值为 cc，所以这里用 CC 来编译 C 源码文件。

下面列出了常见的部分语言程序的隐含规则。

- C 程序

"x.o"的生成依赖于"x.c"，生成 x.o 的命令为：

```
"$(CC) -c $(CPPFLAGS) $(CFLAGS)"。
```

- C++程序

"x.o"的生成依赖于"x.cc"或者"x.C"，生成 x.o 的命令为：

```
"$(CXX) -c $(CPPFLAGS) $(CFLAGS)"
```

- Pascal 程序

"x.o"的生成依赖于"x.p"，生成 x.o 的命令为：

```
"$(PC) -c $(PFLAGS)"。
```

8.1.8　自动化变量

make 还支持一种自动化变量，此变量代表一组文件名，无论是目标文件名，还是依赖文件名，此变量值的范围属于这组文件名集合，也就是说，自动化变量相当于对文件名集合循环遍历一遍。对于不同的文件名集合，有不同的自动化变量名，下面列举一些。

$@，表示规则中的目标文件名集合，如果存在多个目标文件，$@则表示其中每一个文件名。助记，'@'很像是 at，aim at，表示瞄准目标。

$<，表示规则中依赖文件中的第 1 个文件。助记，'<'很像是集合的最左边，也就是第 1 个。

$^，表示规则中所有依赖文件的集合，如果集合中有重复的文件，$^会自动去重。助记，'^'很像从上往下罩的动作，能罩住很大的范围，所以称为集合。

$?，表示规则中，所有比目标文件 mtime 更新的依赖文件集合。助记，'?'表示疑问，make 最大的疑问就是依赖文件的 mtime 是否比目标文件的 mtime 要新。

举个例子，更改 makefile 如下。

```
1 test2.o:test2.c
2            gcc -c -o test2.o test2.c
3 test1.o:test1.c
4            gcc -c -o test1.o test1.c
5 objfiles = test1.o test2.o
6 test.bin:$(objfiles)
7            gcc -o $@ $^
8 all:test.bin
9            @echo "compile done"
```

我们在第 7 行用$@代替了 test.bin，用$^代替了所有依赖文件。所以第 7 行就相当于 gcc –o test.bin test1.o test2.o。执行 make all 后，编译过程正常，test.bin 运行也正常，和之前的图类似，所以不再占版面贴图了。

8.1.9 模式规则

模式，即 pattern，其实就是指字符串模（mú，二声）子，正则表达式中用此概念表示字符或字符串匹配，把符合此模子的字符串找出来，make 中也支持这种字符串匹配用法。

%用来匹配任意多个非空字符。比如%.o 代表所有以.o 为结尾的文件，g%s.o 是以字符 g 开头的所有以.o 为结尾的文件，make 会拿这个字符串模式去文件系统上查找文件，默认为当前路径下。

%通常用在规则中的目标文件中，以用来匹配所有目标文件，%也可以用在规则中的依赖文件中，因为目标文件才是要生成的文件，所以当%用在依赖文件中时，其所匹配的文件名要以目标文件为准。拿%.o:%.c 为例，假如用%.o 匹配到了目标文件 a.o 和 b.o，那么依赖文件中的%.c 将分别匹配到 a.c 和 b.c。

举个例子，现将 makefile 更新如下。

```
1 %.o:%.c
2             gcc -c -o $@ $^
3 objfiles = test1.o test2.o
4 test.bin:$(objfiles)
5             gcc -o $@ $^
6 all:test.bin
7             @echo "compile done"
```

相比上一个 makefile，这个版本中修改了前 2 行，在规则的目标文件中用%.o 匹配所有的.o 文件，当然，目前当前目录下也就是 test1.o 和 test2.o。在依赖文件中用%.c 匹配所有“合适”的.c 文件，这个“合适”是指：要以目标文件中%所匹配到的 test1 和 test2 为主，也就是会匹配到 test1.c 和 test2.c，即使当前目标下还有其他.c 文件也不会受影响匹配结果。

为了演示，我在当前目录下生成了 1.c、2.c 和 3.c 这三个干扰文件，看看 makefile 规则中的%.c 是否会受此影响而出错，如图 8-14 所示。

```
[work@localhost new]$ ls
1.c  2.c  3.c  makefile  my_makefile  test1.c  test2.c
[work@localhost new]$ make all
gcc -c -o test1.o test1.c
gcc -c -o test2.o test2.c
gcc -o test.bin test1.o test2.o
compile done
[work@localhost new]$ ./test.bin
hello,world
[work@localhost new]$ ls
1.c  2.c  3.c  makefile  my_makefile  test1.c  test1.o  test2.c  test2.o  test.bin
[work@localhost new]$
```

▲图 8-14 模式匹配

您看，在图中先用 ls 命令查看下当前目录下的文件，确实多了上面所说的干扰文件：[123].c。

执行 make all 后，make 程序的输出是正常的，test.bin 执行结果也是对的。最后又用了 ls 命令查看当前目录下的文件，一切正常，并未生成多余的文件。

今后我们将有 makefile 来编译程序了，有关 makefile 的内容还是挺多的，比如条件判断、变量操作、内嵌函数等内容咱们还没讲，不过大伙儿放心，没讲的都是咱们用不到的。还是那句话，如果您有兴趣的话还是自己学习下吧，这里就不占用大伙儿的时间了，咱们把时间放在后面的内核代码上。

大伙儿辛苦了，下节再见。

8.2 实现 assert 断言

随着模块越来越多，程序出错的概率将越来越大，为了方便调试，一个好的习惯是在程序中的关键部分设置“哨兵”，让它来监督数据的正确性。这一节我们将介绍断言的实现。

8.2.1 实现开、关中断的函数

断言是什么？其实就是咱们在 C 语言中学过的 assert，想当初刚接触此概念时，我还不清楚为什么把

它称为“断言”。

断，即断定，言，即说，所以断言在表面的意思是断定地说，我断定它一定会怎样怎样。程序中断言的意思也是如此，程序员断定程序运行在此处时，某数据的值一定为多少多少。

在我们系统中，我们实现两种断言，一种是为内核系统使用的 ASSERT，另一种是为用户进程使用的 assert，用户进程离现在还早，咱们本节先实现专供内核使用的 ASSERT。

一方面，当内核运行中出现问题时，多属于严重的错误，着实没必要再运行下去了。另一方面，断言在输出报错信息时，屏幕输出不应该被其他进程干扰，这样咱们才能专注于报错信息。综上两点原因，ASSERT 排查出错误后，最好在关中断的情况下打印报错信息。

内核运行时，为了通过时钟中断定时调度其他任务，大部分情况下中断是打开的，如何在开中断的情况下把中断关闭呢？这就是本节要解决的问题了，咱们马上实现两个自由开关中断的函数。

之前咱们已经建立好了文件 interrupt.c，现在咱们要去更新它了，在原有的基础上，加入了以下内容，大伙儿请见代码 8-1。

代码 8-1　（project/c8/a/kernel/interrupt.c）

```
…略
14 #define EFLAGS_IF   0x00000200       // eflags 寄存器中的 if 位为 1
15 #define GET_EFLAGS(EFLAG_VAR) asm volatile("pushfl; popl %0" : "=g" (EFLAG_VAR))
16
…略
118 /* 开中断并返回开中断前的状态*/
119 enum intr_status intr_enable() {
120    enum intr_status old_status;
121    if (INTR_ON == intr_get_status()) {
122        old_status = INTR_ON;
123        return old_status;
124    } else {
125        old_status = INTR_OFF;
126        asm volatile("sti");      // 开中断，sti 指令将 IF 位置 1
127        return old_status;
128    }
129 }
130
131 /* 关中断，并且返回关中断前的状态 */
132 enum intr_status intr_disable() {
133    enum intr_status old_status;
134    if (INTR_ON == intr_get_status()) {
135        old_status = INTR_ON;
136        asm volatile("cli" : : : "memory"); // 关中断，cli 指令将 IF 位置 0
137        return old_status;
138     } else {
139         old_status = INTR_OFF;
140         return old_status;
141     }
142 }
143
144 /* 将中断状态设置为 status */
145 enum intr_status intr_set_status(enum intr_status status) {
146    return status & INTR_ON ? intr_enable() : intr_disable();
147 }
148
149 /* 获取当前中断状态 */
150 enum intr_status intr_get_status() {
151    uint32_t eflags = 0;
152    GET_EFLAGS(eflags);
153    return (EFLAGS_IF & eflags) ? INTR_ON : INTR_OFF;
154 }
```

代码 8-1 中的内容是在上一版本的基础上新增的部分。

第 14～15 行定义了两个宏，用来获取中断状态。

其中第 14 行的 EFLAGS_IF 表示开中断时 eflags 寄存器中的 IF 的值，由于 IF 位于 eflags 中的第 9 位，故 EFLAGS_IF 的值为 0x00000200。

第 15 行的宏 GET_EFLAGS 用来获取 eflags 寄存器的值。它就是段内嵌汇编代码，其中 EFLAG_VAR 是 C 代码中用来存储 eflags 值的变量，它用寄存器约束 g 来约束 EFLAG_VAR 可以放在内存中或寄存器中，之后用 pushfl 将 eflags 寄存器的值压入栈，然后再用 popl 指令将其弹出到与 EFLAG_VAR 关联的约束中，最后 C 变量 EFLAG_VAR 获得 eflags 的值。

为了管理中断状态，在头文件 interrupt.h 中定义了 enum intr_status 枚举结构，它在代码 8-2 中列出，心急的朋友可以先去看看。该结构很简单，就是 INTR_OFF 值为 0 表示关中断，INTR_ON 值为 1 表示开中断。

先看看第 150 行定义的 intr_get_status 函数，它的作用是获取当前的中断状态。它就是先利用了宏 GET_EFLAGS 来获取 eflags 寄存器的值并存到第 151 行的变量 eflags 中。接下来再利用 EFLAGS_IF 与变量 eflags 的值进行按位与运算，判断变量 eflags 中 IF 位的值是否为 1，从而返回不同的中断状态，即 INTR_ON 或 INTR_OFF。

再回来看第 119 行定义的函数 intr_enable，它的功能是把中断打开，这就是所谓的“开中断”（初次接触这个词时觉得有些高大上），再把执行开中断前的中断状态返回。开中断的原理就是执行 sti 指令将 eflags 中的 IF 位置 1。此函数先调用 intr_get_status 函数获取当前的中断状态，接下来判断：如果当前已经是开中断的状态，那就直接返回，无需再执行一次汇编指令 sti。如果当前是关中断状态，就执行内联汇编 asm volatile("sti")，通过 sti 指令将中断打开。最后，无论程序走哪个分支，都要将操作前的中断状态 old_status 返回。也许您考虑应该把返回指令“return old_status”放在分支判断的外层，也就是放在函数结束前，这样两个分支就共同使用这一行代码了。其实也未尝不可，只是每个分支中都有 return 语句时，能够避免将 C 编译为汇编代码时因为共用一行代码而额外添加 jmp 语句，虽然程序因此而大了一点，但也因此而快了一点，空间换时间。

接下来是第 132 行定义的函数 intr_disable，它的功能是把中断关闭，就是关中断。关中断的原理就是执行 cli 指令将 eflags 中的 IF 位置 0。此函数的原理也是先通过 intr_get_status 获取当前的中断状态，若当前已经是关中断状态，则直接把 INTR_OFF 返回，否则就通过内联汇编“asm volatile("cli" : : : "memory")”将中断打开，然后返回旧中断状态。

第 145 行定义了函数 intr_set_status，它的作用是把中断设置为参数 status 的状态，参数 status 的值通常是调用 intr_disable 和 intr_enable 之后的返回值 old_status，故一般情况下 intr_set_status 用来配合 intr_disable 和 intr_enable，以恢复之前的中断状态。

有关代码 8-1 的介绍就这么多了，现在看下头文件 interrupt.h，请见代码 8-2。

代码 8-2　（project/c8/a/kernel/interrupt.h）

```
 1 #ifndef __KERNEL_INTERRUPT_H
 2 #define __KERNEL_INTERRUPT_H
 3 #include "stdint.h"
 4 typedef void* intr_handler;
 5 void idt_init(void);
 6
 7 /* 定义中断的两种状态:
 8  * INTR_OFF 值为 0，表示关中断
 9  * INTR_ON 值为 1，表示开中断 */
10 enum intr_status {          // 中断状态
11    INTR_OFF,                // 中断关闭
12    INTR_ON                  // 中断打开
13 };
14
15 enum intr_status intr_get_status(void);
16 enum intr_status intr_set_status (enum intr_status);
17 enum intr_status intr_enable (void);
18 enum intr_status intr_disable (void);
19 #endif
```

头文件中的内容很少，就是之前说过的枚举类型 enum intr_status，它用来管理中断，定义了中断的两种状态。INTR_OFF 表示中断关闭，其值为 0。INTR_ON 表示中断打开，基值为 1。

在文件尾是代码 8-1 中介绍的四个函数的声明。

好啦，本节就到这里，下一节咱们实现 ASSERT。

8.2.2　实现 ASSERT

ASSERT 是用来辅助程序调试的，所以通常是用在开发阶段。如果程序中的某些地方会莫名其妙地出错，而我们又无法短时间内将其排查出来，这时我们可以在程序中安排个“哨兵”，这个哨兵就是 ASSERT。我们把程序该有的条件状态传给它，让它帮咱们监督此条件，一旦条件不符合就会报错并将程序挂起。

我们在 C 语言中这样使用 ASSERT：

```
ASSERT (条件表达式);
```

括号中的条件表达式就是上面所说的条件状态。

在 C 语言中 ASSERT 是用宏来定义的，其原理是判断传给 ASSERT 的表达式是否成立，若表达式成立则什么都不做，否则打印出错信息并停止执行，我们仿照它来实现自己的版本。

本节将利用上一节的成果 intr_disable 函数辅助实现 ASSERT，请见代码 8-3。

代码 8-3　（project/c8/a/kernel/debug.h）

```
 1 #ifndef __KERNEL_DEBUG_H
 2 #define __KERNEL_DEBUG_H
 3 void panic_spin(char* filename, int line, const char* func, const char* condition);
 4
 5 /****************************  __VA_ARGS__  *******************************
 6  * __VA_ARGS__ 是预处理器所支持的专用标识符。
 7  * 代表所有与省略号相对应的参数。
 8  * "..."表示定义的宏其参数可变。*/
 9 #define PANIC(...) panic_spin (__FILE__, __LINE__, __func__, __VA_ARGS__)
10  /***********************************************************************/
11
12 #ifdef NDEBUG
13     #define ASSERT(CONDITION) ((void)0)
14 #else
15 #defineASSERT(CONDITION)\
16 if(CONDITION){}else{                     \
17     /* 符号#让编译器将宏的参数转化为字符串字面量 */  \
18      PANIC(#CONDITION);  \
19}
20 #endif /*__NDEBUG */
21
22 #endif /*__KERNEL_DEBUG_H*/
```

代码 8-3 是咱们此次新创建的文件 debug.h，它只是实现 ASSERT 的一部分，在文件开头的 panic_spin 定义在 debug.o 中。

第 12～20 行貌似都是在定义 ASSERT，其实真正有效定义的部分只是第 15 行。

第 15 行用#define 定义 ASSERT(CONDITION)，由于定义的是多行宏，所以各行结尾用反斜杠来续行。

第 16 行判断 CONDITION 条件为真时，其后的大括号中为空，即如前所述，什么都不做。否则条件为假时，在第 18 行调用另外一个宏 PANIC(#CONDITION)，此宏是 ASSERT 采取行动的部分，大伙儿暂且移步到其定义所在的第 9 行。

```
“#define PANIC(...)panic_spin (__FILE__, __LINE__, __func__, __VA_ARGS__)”
```

上面的预处理命令 define 是将 PANIC(...) 定义为 panic_spin 函数，此函数定义在 debug.o 中，一会儿咱们再看它。

大伙儿一定早已注意到了，PANIC 后面是（...），我们知道括号中应该是形参，其实这是 C 预处理器所支持的一种用法，它允许宏支持个数不固定的参数，"..."表示所定义的宏其参数可变，术语为参数个数可变的宏，只要括号中用“…”来占位，就表示此宏的参数个数不固定。悄悄说下，printf 也支持参数个数可变，所以“参数个数可变的宏”和 printf 的声明是一样的：“extern int printf (__const char *__restrict __format, ...)”，您看了 printf 的声明后，是不是更容易接受宏的变参形式了？

参数个数既然不固定，那该如何引用它们呢？

预处理器为此专门提供了一个标识符__VA_ARGS__，它只允许在具有可变参数的宏替换列表中出现，它代表所有与省略号“…”相对应的参数。该参数至少有一个，但可以为空。

所以，我们传给 panic_spin 的其中一个参数是__VA_ARGS__。同样作为参数的还有__FILE__，__LINE__，__func__，这三个是预定义的宏，分别表示被编译的文件名、被编译文件中的行号、被编译的函数名。

咱们再看一下第 18 行，调用 PANIC 的形式为 PANIC(#CONDITION)，即形参为#CONDITION，其中字符'#'的作用是让预处理器把 CONDITION 转换成字符串常量。比如 CONDITION 若为 var != 0，#CONDITION 的效果是变成了字符串“var != 0”。

于是，传给 panic_spin 函数的第 4 个参数__VA_ARGS__，实际类型为字符串指针。

再给大伙儿说一下第 12 行的#ifdef NDEBUG 是怎么回事。

话说 ASSERT 是在调试过程中使用的，宏毕竟是一段代码，经预处理器展开后，调用宏的地方越多，程序体积越大，所以执行得越慢。所以，当我们不再调试时，就应该让此宏失效，这样编译出来的程序才比较小，运行会稍快一些。

话虽如此，但宏已经写在程序中了，怎样让宏在程序中消失呢？难道要写个删除文件行脚本吗？很多流文本命令都可以做这个，比如著名的 sed。写脚本太麻烦了，有没有更好的办法呢？这里可利用#define 宏定义的功能让宏等于空值，这就是在第 13 行所写的#define ASSERT(CONDITION) ((void)0) 的作用，让此 ASSERT 成为空 0，也就是什么都不是，这样就相当于删除了 ASSERT。

宏是预处理器提供的功能，是在预处理阶段处理的，所以让其为空的条件也得在预处理阶段判断才行。在第 12 行我们给出了让宏等于空的条件，用预处理指令#ifdef 判断，如果定义了宏 NDEBUG，就执行上面说过的第 13 行，使 ASSERT 等于 (void)0。此宏 NDEBUG 可以在 gcc 编译时指定，方法很简单，只要用 gcc 的参数-D 来定义 NDEBUG 就行了，如 gcc–DNDEBUG，不过我们通常将“–DNDEBUG”定义在 makefile 中。

好啦，现在再给大伙儿看下 debug.c 中的 panic_spin，见代码 8-4。

代码 8-4 （project/c8/a/kernel/debug.h）

```
 1 #include "debug.h"
 2 #include "print.h"
 3 #include "interrupt.h"
 4
 5 /* 打印文件名、行号、函数名、条件并使程序悬停 */
 6 void panic_spin(char* filename,       \
 7                   int line,           \
 8                   const char* func,     \
 9                   const char* condition) \
10 {
11      intr_disable();     // 因为有时候会单独调用 panic_spin
                            // 所以在此处关中断
12      put_str("\n\n\n!!!!! error !!!!!\n");
13      put_str("filename:");put_str(filename);put_str("\n");
14      put_str("line:0x");put_int(line);put_str("\n");
15      put_str("function:");put_str((char*)func);put_str("\n");
16      put_str("condition:");put_str((char*)condition);put_str("\n");
17      while(1);
18 }
```

代码 8-4 中定义了 panic_spin 函数，除参数 line 外，其他三个参数都是字符串指针，这也证明了代码 8-3 中的 PANIC(#CONDITION)，此#CONDITION 确实是字符串。

代码也比较简单，执行 intr_disable 把中断关闭后，再打印调度相关的信息，之后就通过 while(1) 死循环悬停在此。

为了测试 ASSERT，咱们得找个地方应用它，咱们在 main.c 中测试一下，请见代码 8-5。

代码 8-5 （project/c8/a/kernel/main.c）

```
1 #include "print.h"
2 #include "init.h"
3 #include "debug.h"
4 int main(void) {
5     put_str("I am kernel\n");
```

```
 6     init_all();
 7     ASSERT(1==2);
 8     while(1);
 9     return 0;
10 }
```

相对于上个版本的 main.c，咱们这里把开中断 sti 的内联汇编去掉了，因为之前是为了演示中断机制才打开中断的，目前暂时将其关闭，等真正时机成熟咱们再打开不迟。

为引用 ASSERT，在第 3 行包含了 debug.h，第 7 行用“ASSERT(1==2)”来测试，很明显 1==2 是不成立的，故此 ASSERT 会开始工作，内部调用 panic_spin 来打印调试信息并使程序悬停。

好啦，代码部分该说的都说了，接下来咱们该编译运行了，您懂的，makefile 该派上用场了。

8.2.3　通过 makefile 来编译

老实说，咱们的 makefile 部分介绍得不多，所以实际所写的 makefile 内容也很简单，现在咱们写一个 makefile 来编译之前的所有文件，见代码 8-6。

代码 8-6　（project/c8/a/makefile）

```
 1 BUILD_DIR = ./build
 2 ENTRY_POINT = 0xc0001500
 3 AS = nasm
 4 CC = gcc
 5 LD = ld
 6 LIB = -I lib/ -I lib/kernel/ -I lib/user/ -I kernel/ -I device/
 7 ASFLAGS = -f elf
 8 CFLAGS = -Wall $(LIB) -c -fno-builtin -W -Wstrict-prototypes \
 9          -Wmissing-prototypes
10 LDFLAGS = -Ttext $(ENTRY_POINT) -e main -Map $(BUILD_DIR)/kernel.map
11 OBJS = $(BUILD_DIR)/main.o $(BUILD_DIR)/init.o$(BUILD_DIR)/interrupt.o \
12        $(BUILD_DIR)/timer.o $(BUILD_DIR)/kernel.o $(BUILD_DIR)/print.o \
13        $(BUILD_DIR)/debug.o
14
15 ##############         c 代码编译         ###############
16 $(BUILD_DIR)/main.o: kernel/main.c lib/kernel/print.h \
17         lib/stdint.h kernel/init.h
18         $(CC) $(CFLAGS) $< -o $@
19
20 $(BUILD_DIR)/init.o: kernel/init.c kernel/init.h lib/kernel/print.h \
21         lib/stdint.h kernel/interrupt.h device/timer.h
22         $(CC) $(CFLAGS) $< -o $@
23
24 $(BUILD_DIR)/interrupt.o: kernel/interrupt.c kernel/interrupt.h \
25         lib/stdint.h kernel/global.h lib/kernel/io.h lib/kernel/print.h
26         $(CC) $(CFLAGS) $< -o $@
27
28 $(BUILD_DIR)/timer.o: device/timer.c device/timer.h lib/stdint.h\
29          lib/kernel/io.h lib/kernel/print.h
30        $(CC) $(CFLAGS) $< -o $@
31
32 $(BUILD_DIR)/debug.o: kernel/debug.c kernel/debug.h \
33         lib/kernel/print.h lib/stdint.h kernel/interrupt.h
34        $(CC) $(CFLAGS) $< -o $@
35
36 ##############    汇编代码编译    ###############
37 $(BUILD_DIR)/kernel.o: kernel/kernel.S
38         $(AS) $(ASFLAGS) $< -o $@
39 $(BUILD_DIR)/print.o: lib/kernel/print.S
40         $(AS) $(ASFLAGS) $< -o $@
41
42 ##############    链接所有目标文件    #############
43 $(BUILD_DIR)/kernel.bin: $(OBJS)
44         $(LD) $(LDFLAGS) $^ -o $@
45
46 .PHONY : mk_dir hd clean all
47
48 mk_dir:
49          if [[ ! -d $(BUILD_DIR) ]];then mkdir $(BUILD_DIR);fi
50
```

```
51 hd:
52          dd if=$(BUILD_DIR)/kernel.bin \
53             of=/home/work/my_workspace/bochs/hd60M.img \
54             bs=512 count=200 seek=9 conv=notrunc
55
56 clean:
57          cd $(BUILD_DIR) && rm -f ./*
58
59 build: $(BUILD_DIR)/kernel.bin
60
61 all: mk_dir build hd
```

首先我必须承认这个 makefile 写得一点都不漂亮，完全是为了短、平、快，这样做比较简单省事，我写起来容易，您也容易看懂。好啦废话不多说了，介绍下主要内容。

第 1 行定义了目录变量，即 BUILD_DIR，它用来存储编译生成的所有目标文件。第 2 行定义的变量是 ENTRY_POINT，其值为 0xc0001500，就是之前 ld 命令中-Ttext 参数的值。

第 3～10 行定义了编译器及编译参数，其中在第 8 行的参数变量 CFLAGS 中定义了-fno-builtin，它是告诉编译器不要采用内部函数，因为咱们在以后实现中会自定义与内部函数同名的函数，如果不添加此选项的话，编译时 gcc 会提示与内部函数冲突。-Wstrict-prototypes 选项要求函数声明中必须有参数类型，否则编译时发出警告。-Wmissing-prototypes 选项要求函数必须有声明，否则编译时发出警告。其他内容不再细说。

第 11 行定义的变量 OBJS 用来存储所有的目标文件名，以后每增加一个目标文件，直接在此变量中增加就行了，此变量用在链接阶段。注意，最好不要用模式规则%.o 来匹配，这样不能保证链接顺序，链接时的目标文件，位置顺序上最好还是调用在前，实现在后。

第 15～44 行是编译 C 程序、汇编代码及链接的过程，在前面介绍 makefile 时已经介绍过类似的用法，不再细述。

在代码的末尾定义了 mk_dir，hd，clean，build 及 all 五个伪目标。

其中，伪目标 mk_dir 用来建立 build 目录，通过第 46 行的 shell 命令“if [[! -d $(BUILD_DIR)]]”来判断 build 目录是否存在，若不存在，则利用 mkdir 命令来创建。

伪目标 hd 是将 build/kernel.bin 写入硬盘，执行 make hd 是将文件写入硬盘。

伪目标 clean 是将 build 目录下的文件清空。为稳妥起见，先成功进入 build 目录后再执行“rm –f ./*”删除此目录下的所有文件，避免错删文件。执行 make clean 将会清空 build 目录下的文件。

伪目标 build 就是编译 kernel.bin，只要执行 make build 就是编译文件。

伪目标 all 是依次执行伪目标 mk_dir build hd。只要执行 make all 便完成了编译到写入硬盘的全过程。

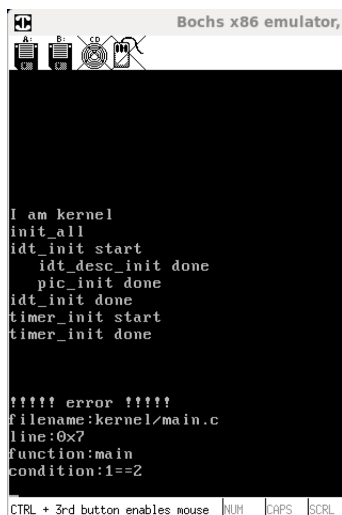
由于此 makefile 还是相当简单的，介绍就到这里，咱们执行下看看，效果如图 8-15 所示。

```
[work@localhost a]$ make all
if [[ ! -d ./build ]];then mkdir ./build;fi
gcc -Wall -I lib/ -I lib/kernel/ -I lib/user/ -I kernel/ -I device/ -c -fno-builtin
-W -Wstrict-prototypes -Wmissing-prototypes -Wsystem-headers kernel/main.c -o build/
main.o
gcc -Wall -I lib/ -I lib/kernel/ -I lib/user/ -I kernel/ -I device/ -c -fno-builtin
-W -Wstrict-prototypes -Wmissing-prototypes -Wsystem-headers kernel/init.c -o build/
init.o
gcc -Wall -I lib/ -I lib/kernel/ -I lib/user/ -I kernel/ -I device/ -c -fno-builtin
-W -Wstrict-prototypes -Wmissing-prototypes -Wsystem-headers kernel/interrupt.c -o b
uild/interrupt.o
gcc -Wall -I lib/ -I lib/kernel/ -I lib/user/ -I kernel/ -I device/ -c -fno-builtin
-W -Wstrict-prototypes -Wmissing-prototypes -Wsystem-headers device/timer.c -o build
/timer.o
nasm -f elf kernel/kernel.S -o build/kernel.o
nasm -f elf lib/kernel/print.S -o build/print.o
gcc -Wall -I lib/ -I lib/kernel/ -I lib/user/ -I kernel/ -I device/ -c -fno-builtin
-W -Wstrict-prototypes -Wmissing-prototypes -Wsystem-headers kernel/debug.c -o build
/debug.o
ld -Ttext 0xc0001500 -e main -Map ./build/kernel.map build/main.o build/init.o build
/interrupt.o build/timer.o build/kernel.o build/print.o build/debug.o -o build/kerne
l.bin
dd if=./build/kernel.bin \
           of=/home/work/my_workspace/bochs/hd60M.img \
           bs=512 count=200 seek=9 conv=notrunc
记录了16+1 的读入
记录了16+1 的写出
8271字节(8.3 kB)已复制，0.000846648 秒，9.8 MB/秒
[work@localhost a]$
```

▲图 8-15 第一个 makefile 执行

您看，屏幕打印了很多信息，这都是命令（gcc、nasm、ld、dd）执行时的输出，从编译、链接到写入硬盘，一气呵成，是不是顿时觉得爽快了太多？看来前期的学习还是值得的。

好，立即在 bochs 中运行检验，效果如图 8-16 所示。



▲图 8-16　ASSERT 效果

您看，正如预期一样，图 8-16 的下面是由“ASSERT(1==2)”打印出来的信息，信息报告了 ASSERT 所在的文件名: kernel/main.c，ASSERT 所在的行号，这里是第 7 行，ASSERT 确实在代码 8-5 的第 7 行，ASSERT 所在的函数：main，ASSERT 中的条件表达式为“1==2”，这里它已经变成了字符串。

到这里，咱们的 ASSERT 实现算是完成了，感谢大家，好累，早点睡。

8.3 实现字符串操作函数

为了将来的开发工作更得心应手，这一节中咱们还是做基础方面的工作，本节打算实现与字符串相关的函数，此类函数以后会被经常用到，还是那句话，磨刀不误砍柴工，咱们要一步一个脚印地走。

在咱们用高级语言编程时，无论是编译型语言，还是脚本语言，都会提供字符串操作函数，而且为了大家使用方便，函数名字基本都与大家所熟识的名字相同或类似，比如获取字符串长度的函数，各种语言版本都类似称为 strlen，必须得遵守约定俗成的名字，否则用户也是不买账的。

咱们以 C 语言为参考，按照 C 代码的字符串函数名编写自己的函数。为此，咱们在 lib 目录下建立了 string.c 文件。大伙放心，这里面定义的每一个函数大家都应该很熟悉，多数都和 C 语言的同名函数功能一样，本节会很轻松。

为方便阅读，将此文件拆分成三部分，请大伙儿参见代码 8-7-1～代码 8-7-3。

代码 8-7-1　（project/c8/b/lib/string.c）

```
 1 #include "string.h"
 2 #include "global.h"
 3 #include "debug.h"
 4
 5 /* 将 dst_起始的 size 个字节置为 value */
 6 void memset(void* dst_, uint8_t value, uint32_t size) {
 7     ASSERT(dst_ != NULL);
 8     uint8_t* dst = (uint8_t*)dst_;
 9     while (size-- > 0)
10         *dst++ = value;
11 }
12
13 /* 将 src_起始的 size 个字节复制到 dst_ */
14 void memcpy(void* dst_, const void* src_, uint32_t size) {
```

```
15     ASSERT(dst_ != NULL && src_ != NULL);
16     uint8_t* dst = dst_;
17     const uint8_t* src = src_;
18     while (size-- > 0)
19         *dst++ = *src++;
20 }
21
22 /* 连续比较以地址 a_和地址 b_开头的 size 个字节，若相等则返回 0，
若 a_大于 b_，返回+1，否则返回-1 */
23 int memcmp(const void* a_, const void* b_, uint32_t size) {
24     const char* a = a_;
25     const char* b = b_;
26     ASSERT(a != NULL || b != NULL);
27     while (size-- > 0) {
28        if(*a != *b) {
29            return *a > *b ? 1 : -1;
30        }
31        a++;
32        b++;
33    }
34    return 0;
35 }
```

memset 函数用于内存区域的数据初始化，原理是逐字节地把 value 写入起始内存地址为 dst_的 size 个空间，在本系统中通常用于内存分配时的数据清 0。

memcpy 函数用于内存数据拷贝，原理是将 src_起始的 size 个字节复制到 dst_，逐字节拷贝。

memcmp 函数用于一段内存数据比较，分别以两个地址 a_和 b_为起始，如果在 size 个字节内，a_中的某个内存字节的数值（或 ASCII 码）大于 b_中同一相对位置的内存数值，此时返回 1，如果这两个地址中，同一位置的所有值都相等，则返回 0，否则返回−1。

代码 8-7-2 （project/c8/b/lib/string.c）

```
37 /* 将字符串从 src_复制到 dst_ */
38 char* strcpy(char* dst_, const char* src_) {
39     ASSERT(dst_ != NULL && src_ != NULL);
40     char* r = dst_;       // 用来返回目的字符串起始地址
41     while((*dst_++ = *src_++));
42     return r;
43 }
44
45 /* 返回字符串长度 */
46 uint32_t strlen(const char* str) {
47     ASSERT(str != NULL);
48     const char* p = str;
49     while(*p++);
50     return (p - str - 1);
51 }
52
53 /* 比较两个字符串，若 a_中的字符大于 b_中的字符返回 1，
相等时返回 0，否则返回-1.  */
54 int8_t strcmp (const char* a, const char* b) {
55     ASSERT(a != NULL && b != NULL);
56     while (*a != 0 && *a == *b) {
57         a++;
58         b++;
59     }
60 /* 如果*a 小于*b 就返回-1，否则就属于*a 大于等于*b 的情况。
在后面的布尔表达式"*a > *b"中，* 若*a 大于*b，表达式就等于 1，
61  否则表达式不成立，也就是布尔值为 0，恰恰表示*a 等于*b */
62    return *a < *b ? -1 : *a > *b;
63 }
64
65 /* 从左到右查找字符串 str 中首次出现字符 ch 的地址*/
66 char* strchr(const char* str, const uint8_t ch) {
67     ASSERT(str != NULL);
68     while (*str != 0) {
69        if (*str == ch) {
70            return (char*)str;   // 需要强制转化成和返回值类型一样
                                   // 否则编译器会报 const 属性丢失
```

```
71          }
72          str++;
73       }
74       return NULL;
75 }
```

strcpy 函数用于把起始于地址 src_的字符串复制到地址 dst_，这和 memcpy 原理相同，只不过 strcpy 以 src_处的字符’0’作为终止条件，memcpy 以拷贝的字节数 size 为终止条件。

strlen 函数用于返回字符串的长度，即字符数。

strcmp 函数用于比较起始地址分别为 a 和 b 的两个字符串是否相等，若 a 中某个字符的 ASCII 值大于 b 中同一相对位置字符的 ASCII 值，此时返回 1，若字符都相同，则返回 0，否则返回−1。同 memcpy 的原理相同，区别就是 strcmp 以地址 a 处的字符串的长度，也就是直到字符 0 为终止条件，memcpy 的终止条件是 size 个比对的字节。

strchr 返回的是字符 ch 在字符串 str 中，从左到右最先出现的所在地址，并不是下标，这一点请注意。

代码 8-7-3　（project/c8/b/lib/string.c）

```
77 /* 从后往前查找字符串 str 中首次出现字符 ch 的地址*/
78 char* strrchr(const char* str, const uint8_t ch) {
79      ASSERT(str != NULL);
80      const char* last_char = NULL;
81 /* 从头到尾遍历一次，若存在 ch 字符，last_char 总是该字符最后一次
   出现在串中的地址(不是下标，是地址)*/
82      while (*str != 0) {
83         if (*str == ch) {
84              last_char = str;
85         }
86         str++;
87     }
88     return (char*)last_char;
89 }
90
91 /* 将字符串 src_拼接到 dst_后，返回拼接的串地址 */
92 char* strcat(char* dst_, const char* src_) {
93      ASSERT(dst_ != NULL && src_ != NULL);
94      char* str = dst_;
95      while (*str++);
96      --str;      // 别看错了，--str 是独立的一句，并不是 while 的循环体
97      while((*str++ = *src_++));  //当*str 被赋值为 0 时
        //也就是表达式不成立，正好添加了字符串结尾的 0
98    return dst_;
99 }
100
101 /* 在字符串 str 中查找字符 ch 出现的次数 */
102 uint32_t strchrs(const char* str, uint8_t ch) {
103      ASSERT(str != NULL);
104      uint32_t ch_cnt = 0;
105      const char* p = str;
106      while(*p != 0) {
107        if (*p == ch) {
108             ch_cnt++;
109        }
110        p++;
111      }
112      return ch_cnt;
113 }
```

strrchr 函数返回的是从后往前查找字符串 str 中首次出现字符 ch 的地址，注意，是字符在字符串中的地址，并不是下标值。此函数虽然是从后往前找，但原理上是通过从前往后（从左到右）的顺序查找的，这样的好处是无需事先知道字符串的结束字符’\0’的地址。

strcat 函数的功能是字符串拼接，将 src_处的字符串接在 dst_的结尾处，并将 dst_返回。实现原理是将 src_处的字符串拷贝到 dst_的结束处。

strchrs 函数用于返回字符 ch 在字符串 str 中出现的次数。

您看，是不是很容易？本节很愉快地结束了。

8.4 位图 bitmap 及其函数的实现

8.4.1 位图简介

位图，也就是 bitmap，广泛用于资源管理，是一种管理资源的方式、手段。“资源”包括很多，比如内存或硬盘，对于此类大容量资源的管理一般都会采用位图的方式。

什么是位图呢？位图包含两个概念：位和图。

位是指 bit，即字节中的位，1 字节中有 8 个位。图是指 map，map 这个词在很久之前就介绍过啦，地图本质上就是映射的意思，映射，即对应关系。综合起来，位图就是用字节中的 1 位来映射其他单位大小的资源，按位与资源之间是一对一的对应关系。

生活中处处有这样一一对应的例子，比如老师上课用于点名的名单，上面每一个名字都代表班里某一个真实的同学，去银行办事也要先排号，这个号也是对应某一个来办事的人。

计算机中为什么要使用位图呢？其实我不说您也明白，不过我还是自由发挥一下吧。

计算机中的一些资源的数量非常庞大，比如内存容量和硬盘空间，发展到今天，它们的容量还是非常可观的。为了有效使用这些资源，必然要涉及到一套管理这些资源的方法。而方法中必然要构建具体的数据结构来存储管理数据，而这些数据结构本身也要占用内存，也就是说，管理资源的方法是有成本的，这个成本当然是越小越好。您想，就拿内存管理来说，总不能用物理内存的一半容量来存储资源管理的相关数据吧。

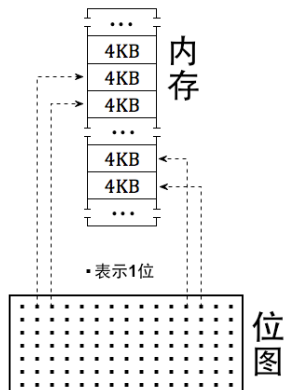
管理结构中的数据也有自己的单位大小，被管理的资源也有自己的单位大小，故有效减少管理成本的方法是使管理结构中的单位达到最小，其所管理资源的单位调整到最大。此类典型的案例有很多，比如一个足球大小的地球仪代表整个地球。

计算机中最小的数据单位是位，于是，用一组二进制位串来管理其他单位大小的资源是很自然的事，这组二进制位中的每一位与其他资源中的数据单位都是一对一的关系，这实际就成了一种映射，即 map，于是这组二进制位就有了更恰当的名字——位图。

既然位图本质上就是一串二进制位，那对于它的实现，用字节型数组还是比较方便的，数组中的每一个元素都是一字节，每 1 字节含有 8 位，因此位图的 1 字节对等表示 8 个资源单位。

位图中的每一位有两种状态，即 0 和 1，所以一般情况下，位图所管理的资源被我们人工划分为两种状态（并不是说资源本身就只有两种状态），状态的定义是人为的，取决于咱们怎样看待它们，取决于管理的内容是什么。位图用 0 和 1 这两种状态反应实际所管理资源的状态，比如位图中的 0 表示该资源未占用，位图中的 1 表示该资源已占用。当然这只是举例而已，具体的意义要随具体的应用而定。

也许我还没把“管理结构的单位大小”和“资源自己的单位大小”说清楚。举个例子来说，若用位图来管理内存，位图中的每一位都将表示实际物理内存中的 4KB，也就是一页，即位图中的一位对应物理内存中的一页，如果某位为 0，表示该位对应的页未分配，可以使用，反之如果某位为 1，表示该位对应的页已经被分配出去了，在将该页回收之前不可再分配。这种对应关系如图 8-17 所示。



▲图 8-17 位图与内存

其中，“管理结构的单位大小”是指位图中的 1 位，也就是图 8-17 位图框中的“黑点”，“资源自己的单位大小”就是指以 4KB 为单位大小的内存，也就是图中每一个 4KB 的小格子。注意，内存本身最小可寻址单位是字节，4KB 是人为划分的内存单位，内存中可没有一个个 4KB 大小的“格子”。之所以这样做，原因就像上面所说的，将所管理资源的单位调整到最大。当然，为了高效管理，这个资源单位并不是越大越好，得方便管理才合适。

以上是我随意发挥的，表达能力有限，希望能帮助那些头一次接触位图的同学。

总结一下，位图相当于一组资源的映射。位图中的每一位和被管理的单位资源都是一对一的关系，故位图主要用于管理容量较大的资源。

8.4.2 位图的定义与实现

上一节中给大伙介绍了位图的概念，不知道那些未接触过位图的同学是否清楚了？不过用文字描述还是有些单薄晦涩，对于抽象的东西多说无益，咱们还是看看实际的位图代码吧。

本节又新增两个文件 bitmap.c 和 bitmap.h，它们位于 lib/kernel/下，咱们先看下 bitmap.h，请见代码 8-8。

代码 8-8 （project/c8/c/lib/kernel/bitmap.h）

```
 1 #ifndef __LIB_KERNEL_BITMAP_H
 2 #define __LIB_KERNEL_BITMAP_H
 3 #include "global.h"
 4 #define BITMAP_MASK 1
 5 struct bitmap {
 6     uint32_t btmp_bytes_len;
 7 /* 在遍历位图时，整体上以字节为单位，细节上是以位为单位，
   所以此处位图的指针必须是单字节 */
 8     uint8_t* bits;
 9 };
10
11 void bitmap_init(struct bitmap* btmp);
12 bool bitmap_scan_test(struct bitmap* btmp, uint32_t bit_idx);
13 int bitmap_scan(struct bitmap* btmp, uint32_t cnt);
14 void bitmap_set(struct bitmap* btmp, uint32_t bit_idx, int8_t value);
15 #endif
```

给大伙儿看 bitmap.h 的目的是想让大伙儿先了解位图的结构 struct bitmap，这样在后面介绍位图操作函数时就更容易理解了。

struct bitmap 中只定义了两个成员：位图的指针 bits 和位图的字节长度 btmp_bytes_len。

前面和大伙儿介绍过，位图可以使用字节型数组来生成，如果位图长度事先已知的话，我们可以在 struct bitmap 中定义一个字节型数组成员来充当位图。当然这也只是愿景，现实中哪有那么多容易的事，我“悲观”的原因是这样的，位图长度取决于所管理资源的大小，其长度不固定，因此不能在位图结构 struct bitmap 中生成固定大小的位图数组。因此一种“乐观”的解决方案是在 struct bitmap 中提供位图的指针，就是 uint8_t* bits。用指针 bits 来记录位图的地址，真正的位图由上一级模块提供，并由上一级模块把位图的地址赋值给 bits。

注意，bits 的类型是 uint8_t*，此类型强调的是字节型指针，最好不要用多字节类型，否则在处理时会复杂。因为我们在遍历位图时先是通过字节来定位某 bit 所在的字节，如果将其设为其他类型，比如 int32，在遍历位图时会产生 4 字节的跳跃，每次处理其中的 bit 时要判断 32 个 bit，不如让数组成员为字节型，这样字节偏移量就是数组元素的索引，定位方便，而且只处理 8 个 bit，因此指针 bits 的类型最好设为单字节大小。

宏 BITMAP_MASK 其值为 1，用来在位图中逐位判断，主要就是通过按位与‘&’来判断相应位是否为 1。

好啦，头文件就这么简单，下面看下位图的具体实现，请大伙儿参见代码 8-9。

代码 8-9 （project/c8/c/lib/kernel/bitmap.c）

```
 1 #include "bitmap.h"
 2 #include "stdint.h"
 3 #include "string.h"
 4 #include "print.h"
 5 #include "interrupt.h"
 6 #include "debug.h"
 7
 8 /* 将位图 btmp 初始化 */
 9 void bitmap_init(struct bitmap* btmp) {
10     memset(btmp->bits, 0, btmp->btmp_bytes_len);
11 }
12
13 /* 判断 bit_idx 位是否为 1，若为 1，则返回 true，否则返回 false */
14 bool bitmap_scan_test(struct bitmap* btmp, uint32_t bit_idx) {
15     uint32_t byte_idx = bit_idx / 8;    // 向下取整用于索引数组下标
```

```
16     uint32_t bit_odd  = bit_idx % 8;    // 取余用于索引数组内的位
17     return (btmp->bits[byte_idx] & (BITMAP_MASK << bit_odd));
18 }
19
20 /* 在位图中申请连续cnt个位，成功，则返回其起始位下标，失败，返回-1 */
21 int bitmap_scan(struct bitmap* btmp, uint32_t cnt) {
22     uint32_t idx_byte = 0;       // 用于记录空闲位所在的字节
23 /* 先逐字节比较，蛮力法 */
24     while (( 0xff == btmp->bits[idx_byte]) && (idx_byte < btmp->btmp_bytes_len)) {
25 /* 1表示该位已分配，若为0xff，则表示该字节内已无空闲位，向下一字节继续找 */
26        idx_byte++;
27     }
28
29     ASSERT(idx_byte < btmp->btmp_bytes_len);
30     if (idx_byte == btmp->btmp_bytes_len) {  // 若该内存池找不到可用空间
31          return -1;
32     }
33
34  /* 若在位图数组范围内的某字节内找到了空闲位
35  *在该字节内逐位比对，返回空闲位的索引。*/
36    int idx_bit = 0;
37  /* 和btmp->bits[idx_byte]这个字节逐位对比 */
38    while ((uint8_t)(BITMAP_MASK << idx_bit) & btmp->bits[idx_byte]) {
39            idx_bit++;
40    }
41
42    int bit_idx_start = idx_byte * 8 + idx_bit;    // 空闲位在位图内的下标
43    if (cnt == 1) {
44        return bit_idx_start;
45    }
46
47    uint32_t bit_left = (btmp->btmp_bytes_len * 8 - bit_idx_start);
   // 记录还有多少位可以判断
48    uint32_t next_bit = bit_idx_start + 1;
49    uint32_t count = 1;          // 用于记录找到的空闲位的个数
50
51    bit_idx_start = -1;          // 先将其置为-1，若找不到连续的位就直接返回
52    while (bit_left-- > 0) {
53       if (!(bitmap_scan_test(btmp, next_bit))) {    // 若next_bit为0
54            count++;
55       } else {
56           count = 0;
57       }
58       if (count == cnt) {          // 若找到连续的cnt个空位
59            bit_idx_start = next_bit - cnt + 1;
60            break;
61       }
62       next_bit++;
63    }
64   return bit_idx_start;
65 }
66
67 /* 将位图btmp的bit_idx位设置为value */
68 void bitmap_set(struct bitmap* btmp, uint32_t bit_idx, int8_t value) {
69     ASSERT((value == 0) || (value == 1));
70     uint32_t byte_idx = bit_idx / 8;    // 向下取整用于索引数组下标
71     uint32_t bit_odd  = bit_idx % 8;    // 取余用于索引数组内的位
72
73 /* 一般都会用个0x1这样的数对字节中的位操作
74  * 将1任意移动后再取反，或者先取反再移位，可用来对位置0操作。*/
75   if (value) {              // 如果value为1
76       btmp->bits[byte_idx] |= (BITMAP_MASK << bit_odd);
77   } else {                 // 若为0
78       btmp->bits[byte_idx] &= ~(BITMAP_MASK << bit_odd);
79   }
80 }
```

给大伙儿简要介绍下 bitmap.c，代码虽然不长，但今后我们所用到的位图操作够用了。

bitmap_init 函数只有一个参数，即位图指针 btmp，此函数功能是初始化位图 btmp，它是用 memset

函数根据位图的字节大小 btmp_bytes_len 将位图的每一个字节用 0 来填充。

bitmap_scan_test 函数接受两个参数，分别是位图指针 btmp 和位索引 bit_idx。其功能是判断位图 btmp 中的第 bit_idx 位是否为 1，若为 1，则返回 true，否则返回 false。此函数是被 bitmap_scan 调用的，当想在位图中获得连续多个可用位时，逐次调用 bitmap_scan_test 依次判断。

bitmap_scan 函数接受两个参数，分别是位图指针 btmp 及位的个数 cnt。此函数的功能是在位图 btmp 中找到连续的 cnt 个可用位，返回起始空闲位下标，若没找到 cnt 个空闲位，返回−1。位图中的位，其值为 0 就表示该位对应的资源可用，故此函数的原理是在 btmp->bits 所指向的位图中先逐字节查找值为 0 的位所在的字节，以该字节为起始，然后逐个判断每一个位，若找到连续的 cnt 值为 0 的空闲位，返回第一个空闲位所在的下标。bitmap_scan 是位图操作的核心方法，其他位图函数也用到了该函数中的方法，下面看其实现。

第 22 行的 idx_byte 用于记录第一个空闲位所在的字节索引，此变量用在下面的 while 循环中，在位图的字节长度 btmp_bytes_len 范围内逐字节查找，若字节的值不为 0xff，就说明该字节里面至少有一个 0，于是循环条件不成立，退出循环。

第 30 行，如果 idx_byte 等于位图的字节长度，表示位图已没有空闲位，直接返回−1。

接下来在该字节内逐位判断，第 36 行定义一个变量 idx_bit，用于记录此字节内第一个空闲位的下标。第 38 行在含有空闲位的字节（idx_byte）内逐个判断所有位，判断的方法是在 while 循环中，用 BITMAP_MASK << idx_bit 将 1 逻辑左移到各位，然后与该字节内的各位进行按位与操作，直到按位与的结果为 0 时退出循环，因此找到空闲位，其下标就记在变量 idx_bit 中。

idx_bit 仅仅是字节内的索引，其值范围是 0～7，我们在第 42 行声明一个变量 bit_idx_start，将 idx_bit 转换成整个位图内的位索引，方法很简单，就是加了个空闲位所在字节的位数：idx_byte * 8。

在第 43 行判断 cnt 的大小，若只想获取一个空闲位，即 cnt 为 1 的话，bit_idx_start 就是最终的结果，直接在第 44 行将其返回。

如果 cnt 大于 1，这说明还要继续在位图中找空闲位，但我们得知道位图中还有多少个剩余位，一定不能访问到位图外的内存。所以在第 47 行，用变量 bit_left 来记录位图内还有多少个位可以判断，它的值等于位图中位的总数量减去第一个空闲位的索引，即 btmp_bytes_len * 8 - bit_idx_start。

第 48 行的 next_bit 用于记录位图中下一个待查找的位，它是相对于整个位图的位下标。目前也不知道 next_bit 位是空闲位（0），还是已经占用（1），一会要传给函数 bitmap_scan_test 去判断。

第 49 行的变量 count 用于记录找到的空闲位的个数，我们不是要找到 cnt 个空闲位吗，当找到的空闲位 count 等于 cnt 时，就表示咱们成功找到了 cnt 个空闲位。

虽然 bit_idx_start 是位图中第一个空闲位的下标，但我们的目的是找到连续的 cnt 个空闲位，若找不到 cnt 个连续空闲位就失败返回。于是在第 51 行，将变量 bit_idx_start 置为−1，即默认情况下找不到 cnt 个空闲位（先悲观着没什么不好，万一有惊喜呢）。

第 52～63 行便开足马力，通过调用 bitmap_scan_test 依次判断下一位 next_bit 是否为 0，即是否为空闲，每判断一位就在第 62 行将 next_bit++。每找到一个连续的空闲位就将 count++，若下一个位不是空闲，就将 count 清 0，从头再来重新找。若发现 count 等于 cnt，说明完成任务，将 bit_idx_start 改为连续 cnt 个空闲位的起始，即“bit_idx_start = next_bit - cnt + 1”。通过 break 退出循环，执行“return bit_idx_start”返回。否则，当 bit_left 递减为 0 时，即表示位图中所有的位都检索过了，于是 while 循环条件不成立，循环退出，此时 bit_idx_start 依然是−1，于是将 bit_idx_start 返回表示失败。至此 bitmap_scan 就介绍完了。

bitmap_set 接受三个参数，位图指针 btmp、位索引 bit_idx、位值 value，函数功能是将位图 btmp 中的第 bit_idx 位设置为 value，其中 bit_idx 为整个位图中的位索引。简要介绍下此函数。

第 69 行的 ASSERT 用来监督 value 的范围，既然是为位设置值，value 要么是 0，要么是 1。

第 70～71 行将 bit_idx 转换成相应的字节 byte_idx 及字节内的偏移 bit_odd。

第 75～79 行是将字节 bits[byte_idx]中的第 bit_odd 位置为 value，也就是将位图中的第 bit_idx 位的值置为 value。对于 value 的值，这里的处理分两种情况。

如果 value 为 1，直接将 BITMAP_MASK（宏值为 1）左移 bit_odd 位后与位图中的字节 bits[byte_idx]

进行按位或运算，这样该字节中的位 bit_odd 便为 1，其他位为原值。

如果 value 为 0，即只将字节 bits[byte_idx] 中的 bit_odd 位置为 0，方法是让某个字节中的 bit_odd 位为 0，其他位为 1，然后将该字节与 bits[byte_idx] 做按位与运算。这样 bits[byte_idx]中的第 bit_odd 位便置为 0，其他位为原值。

好啦，有关位图操作的函数到这里就介绍完了，本节到此结束，大伙儿辛苦啦。

8.5 内存管理系统

程序即是一堆指令的集合，而内存是程序的舞台。要想让程序在计算机中运行，必须要将其加载到内存中才可以，原因是处理器被设计成到内存中取指令，这就是处理器的代码段寄存器 CS 和指令指针寄存器 EIP 中存储的是指令所在的内存地址的原因。

用户程序所占用的内存空间是由操作系统分配的，内存是如何分配的并且该给用户进程分配多少字节呢？这就是咱们要解决的问题。

所以，从现在起，咱们要循序渐进地实现内存管理系统，一直到函数 malloc 和 free 的完成。

当然，饭要一口一口地吃，做任何事情都不能一蹴而就，本节咱们先实现内存管理系统的基础部分，然后再完成简单的内存分配。

8.5.1 内存池规划

通常情况下，我们所说的地址都是指内存地址。虽然端口和扇区也都是通过地址定位，但为了与内存地址区分，它们的地址亦分别称为端口号和逻辑扇区号，所以，地址和内存几乎被我们一视同仁，地址和内存，它们虽然一一对应，但却是两套不同的资源。

我们知道，内核和用户进程分别运行在自己的地址空间，在实模式下，程序中的地址就等于物理地址，实实在在，这没什么好说的。在保护模式下，程序地址变成了虚拟地址，虚拟地址对应的物理地址是由分页机制做的映射。因此，在分页机制下有了虚拟、物理这两种地址，操作系统有责任把这两种地址分别管理，并通过页表将这两类地址关联。我们本节所讨论的就是有关这两类地址的内存池规划问题。

什么是内存池？本文所讨论的内存池，其实称为内存地址池更为恰当。

池，意为池塘，也就是水源仓库，它起到水源存储、集中管理的作用，需要水的时候直接从池中取出即可。内存地址池的概念是将可用的内存地址集中放到一个“池子”中，需要的时候直接从里面取出，用完后再放回去。由于在分页机制下有了虚拟地址和物理地址，为了有效地管理它们，我们需要创建虚拟内存地址池和物理内存地址池。

咱们先讨论下如何规划物理内存池。

不管怎么说，内核和用户进程肯定都要运行在物理内存之中。问题来了，哪些物理内存用来运行内核，哪些物理内存用来运行用户进程呢？咱们先看看物理内存的规划。

一种可行的方案是将物理内存划分成两部分，一部分只用来运行内核，另一部分只用来运行用户进程，将内存规划出不同的部分，专项专用。

操作系统为了能够正常运行，不能用户进程申请多少内存就分配多少，必须得给自己预留出足够的内存才行，否则有可能会出现因为物理内存不足，导致内核自己都无法正常运行、自身难保的现象。

基于这个原因，我们把物理内存分成两个内存池，一部分称为用户物理内存池，此内存池中的物理内存只用来分配给用户进程。另一部分就是内核物理内存池，此内存池中的物理内存只给操作系统使用。

即使是从池塘中取水，每次也只是取出一小部分，要么用水桶盛水，要么用盆盛水，总之每次取水都会有一个容量单位，不会一次就把整个池塘的水都取走，当然连续多次取水还是有可能的。从内存池中获取内存资源也是一样的，内存池中的内存也得按单位大小来获取，这个单位大小是 4KB，称为页，故，内存池中管理的是一个个大小为 4KB 的内存块，从内存池中获取的内存大小至少为 4KB 或者为 4KB 的倍数（以后咱们会实现更细粒度的内存管理，但是，这依然不是直接从内存池中获取，它需要另外一种管理结构，这是后话）。

为了方便实现，咱们把这两个内存池的大小设为一致，即各占一半的物理内存，如图 8-18 所示。

当用户内存池中的内存都被用户进程耗尽时，不再向内核内存池申请，而是返回信息"内存不足"，拒绝请求。

下面再讨论下虚拟内存地址池。

先来点前情提要，就当是与大伙儿回顾下。在分页机制下程序中的地址都是虚拟地址，虚拟地址的范围取决于地址总线的宽度，咱们是在 32 位环境下，所以虚拟地址空间为 4GB。除了地址空间比较大以外，分页机制的另一个好处是每个任务都有自己的 4GB 虚拟地址空间，也就是各程序中的虚拟地址不会与其他程序冲突，都可以为相同的虚拟地址，不仅用户进程是这样，内核也是。程序中的地址是由链接器在链接过程中分配的，分配之后就不会再变了，运行时按部就班地送上处理器的 CS 和 EIP 即可。

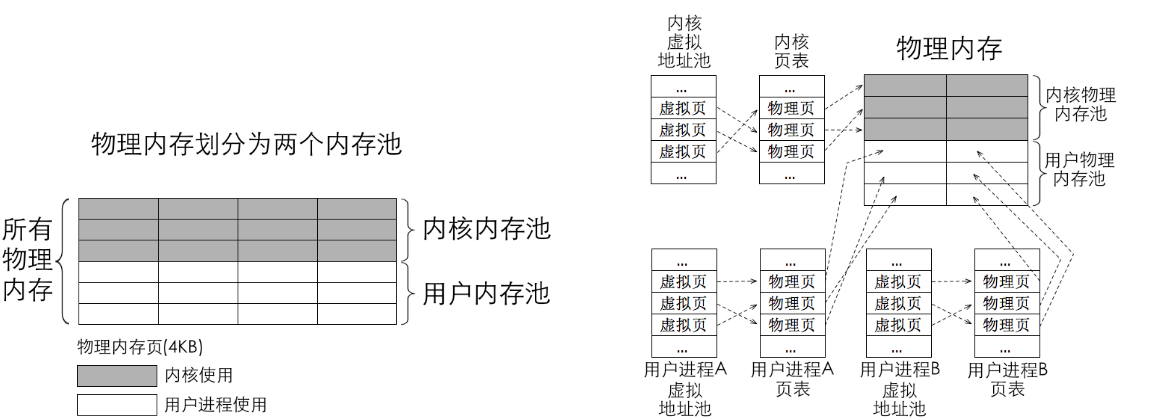
但程序（进程、内核线程）在运行过程中也有申请内存的需求，这种动态申请内存一般是指在堆中申请内存，操作系统接受申请后，为进程或内核自己在堆中选择一空闲的虚拟地址，并且找个空闲的物理地址作为此虚拟地址的映射，之后把这个虚拟地址返回给程序。那么问题又来了，哪些虚拟地址是空闲的？如何跟踪它们的？

对于所有任务（包括用户进程、内核）来说，它们都有各自 4GB 的虚拟地址空间，因此需要为所有任务都维护它们自己的虚拟地址池，即一个任务一个。

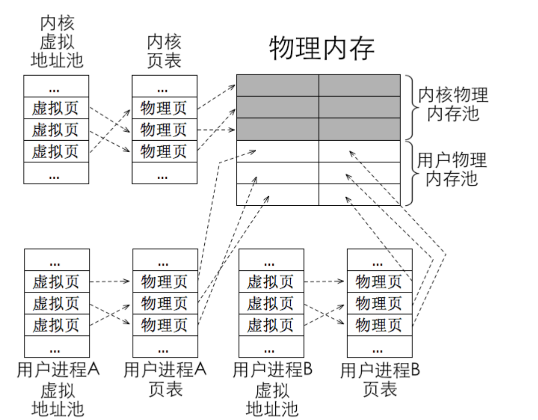
内核为完成某项工作，也需要申请内存，当然它绝对有能力不通过内存管理系统申请内存，先斩后奏，或者奏都不奏一声，直接拿来就用。当然，这种"王者之风"显然不是那么和谐，我们让内核也通过内存管理系统申请内存，为此，它也要有个虚拟地址池，当它申请内存时，从内核自己的虚拟地址池中分配虚拟地址，再从内核物理内存池（内核专用）中分配物理内存，然后在内核自己的页表将这两种地址建立好映射关系。

对用户进程来说，它向内存管理系统，即操作系统，申请内存时，操作系统先从用户进程自己的虚拟地址池中分配空闲虚拟地址，然后再从用户物理内存池（所有用户进程共享）中分配空闲的物理内存，然后在该用户进程自己的页表将这两种地址建立好映射关系。

为方便管理，虚拟地址池中的地址单位也是 4KB，这样虚拟地址便于和物理地址做完整页的映射。有了虚拟地址池和物理地址池后，它们的关系如图 8-19 所示。



▲图 8-18　内存池示意



▲图 8-19　虚拟地址池与物理地址池

无论说得再怎么细致都不如自己动手做一遍来得清楚，咱们马上动手实践。

本节在 kernel 目录下新建了两个文件 memory.h 和 memory.c，有关内存管理的代码都写在其中。在头文件中定义了虚拟地址结构，咱们一睹为快，如代码 8-10 所示。

代码 8-10　（c8/d/kernel/memory.h）

```
1 #ifndef __KERNEL_MEMORY_H
2 #define __KERNEL_MEMORY_H
3 #include "stdint.h"
4 #include "bitmap.h"
5
```

```
 6 /* 虚拟地址池，用于虚拟地址管理 */
 7 struct virtual_addr {
 8     struct bitmap vaddr_bitmap; // 虚拟地址用到的位图结构
 9     uint32_t vaddr_start;       // 虚拟地址起始地址
10 };
11
12 extern struct pool kernel_pool, user_pool;
13 void mem_init(void);
14 #endif
```

memory.h 比较简单，主要就是定义了 struct virtual_addr，此结构就是虚拟地址池，用于虚拟地址管理。

坦白说，struct virtual_addr 是不必要的，对于虚拟地址的管理，在页表中完全可以直接把虚拟地址和物理地址一一对应，这样实现还更简单，一对一映射是最方便的。不过，即使没学过操作系统课程，大伙儿也都有所了解，虚拟地址与物理地址是两种不同的地址概念，它们是通过分页机制，用页表的映射关系关联到一块的，也就是说，虚拟地址和物理地址是各自独立不相关的。其中虚拟地址是逻辑上的，通常可指程序中的地址，其具有连续性。物理地址是指分布在物理内存中的真实地址，可以连续，也可以不连续，为了演示分页机制是如何将这两种独立不相关的地址关联到一起的，特意加了此结构体用于管理虚拟地址。

继续说代码，struct virtual_addr 包含两个成员，一个是 vaddr_bitmap，它的类型是位图结构体 struct bitmap，用来以页为单位管理虚拟地址的分配情况，对，您没看错，虚拟地址也要分配。虽然多个进程可以拥有相同的虚拟地址，但究其原因，是因为这些虚拟地址所对应的物理地址是不同的。但是，在同一个进程内的虚拟地址必然是唯一的，这通常是由链接器为其分配的，由链接器负责虚拟地址（程序内地址）的唯一性。但进程在运行时可以动态从堆中申请内存，系统为其分配的虚拟地址也属于此进程的虚拟地址空间，也必须要保证虚拟地址的唯一性，所以，用位图来记录虚拟地址的分配情况。

vaddr_start 用来记录虚拟地址的起始值，咱们将来在分配虚拟地址时，将以这个地址为起始分配。其他的部分是一些声明，它们都在 memory.c 中有具体的实现，在此不解释了。

下面咱们看看 memory.c 的具体代码。

为方便大伙儿阅读，我已把代码划分成三部分贴出来。大伙先请看代码 8-11-1。

代码 8-11-1 （project/c8/d/kernel/memory.c）

```
 1 #include "memory.h"
 2 #include "stdint.h"
 3 #include "print.h"
 4
 5 #define PG_SIZE 4096
 6
 7 /***********************  位图地址 ****************************
 8  * 因为 0xc009f000 是内核主线程栈顶，0xc009e000 是内核主线程的 pcb。
 9 * 一个页框大小的位图可表示 128MB 内存，位图位置安排在地址 0xc009a000，
10  * 这样本系统最大支持 4 个页框的位图，即 512MB */
11 #define MEM_BITMAP_BASE 0xc009a000
12 /*************************************************************/
13
14 /* 0xc0000000 是内核从虚拟地址 3G 起。
0x100000 意指跨过低端 1MB 内存，使虚拟地址在逻辑上连续 */
15 #define K_HEAP_START 0xc0100000
16
17 /* 内存池结构，生成两个实例用于管理内核内存池和用户内存池 */
18 struct pool {
19     struct bitmap pool_bitmap; //本内存池用到的位图结构，用于管理物理内存
20     uint32_t phy_addr_start;    // 本内存池所管理物理内存的起始地址
21     uint32_t pool_size;        // 本内存池字节容量
22 };
23
24 struct pool kernel_pool, user_pool; // 生成内核内存池和用户内存池
25 struct virtual_addr kernel_vaddr;   // 此结构用来给内核分配虚拟地址
26
27 /* 初始化内存池 */
28 static void mem_pool_init(uint32_t all_mem) {
29    put_str("   mem_pool_init start\n");
30    uint32_t page_table_size = PG_SIZE * 256;
```

```
      // 页表大小 = 1 页的页目录表 + 第 0 和第 768 个页目录项指向同一个页表 +
31  // 第 769～1022 个页目录项共指向 254 个页表，共 256 个页框

32    uint32_t used_mem = page_table_size + 0x100000;
                              // 0x100000 为低端 1MB 内存
33    uint32_t free_mem = all_mem - used_mem;
34    uint16_t all_free_pages = free_mem / PG_SIZE;
// 1 页为 4KB，不管总内存是不是 4k 的倍数
35          // 对于以页为单位的内存分配策略，不足 1 页的内存不用考虑了
```

第 5 行定义了宏 PG_SIZE，用以表示页的尺寸，其值为 4096，即 4KB。

第 11 行定义了宏 MEM_BITMAP_BASE，用以表示内存位图的基址，其值为 0xc009a000，为什么要选择这个数呢？

这其实和进程 PCB 或线程 TCB（PCB 程序控制块，TCB 线程控制块）结构有一定关系。为方便讨论我们暂且称之为 PCB 吧。将来我们所实现的 PCB 要占用 1 页内存，即 PCB 要占用 4KB 大小的内存空间，不过要注意的是 PCB 所占用的内存必须是自然页，自然页就是页的起始地址必须是 0xXXXXX000，终止地址必须是 0xXXXXXfff。也就是不能跨页占用，PCB 必须是完整、单独地占用一个物理页框。顺便说一句，PCB 是进程或线程的“身份证”，任何一个进程都包含一个 PCB 结构。

为方便叙述 PCB 的结构，假如 PCB 的地址是 0xXXXXX000。PCB 结构是这样的，在 PCB 的最低处 0xXXXXX000 以上存储的是进程或线程的信息，这包括 pid、进程状态等（以后在介绍线程时再跟大伙儿细说）。PCB 的最高处 0xXXXXXfff 以下用于进程或线程在 0 特权级下所使用的栈。

因为压栈操作的原理是栈指针 esp 先自减，然后再往自减后的地址处存储数据，故，任何进程或线程初始的栈顶便是此页框的最顶端+1，即下一个页框的起始处，也就是 0xXXXXXfff + 1。

程序都有个主线程，咱们的内核也是一样，这个主线程就是指正式进入内核时所运行的程序，其实就是 main 线程。main 线程一直存在，它调用了 init_all 来做各种初始化的工作，将来还要初始化线程等。您看，它默默地为咱们做了这么多，咱们得给它个名份，也就是说，它必须也要有个 PCB。

其实，我们早已经为 main 预留了 PCB 的空间，正如咱们在 loader.S 中所做的，在进入内核之前，通过 mov esp, 0xc009f000 将内核所使用的栈顶指向 0xc009f000，因此您肯定知道了将来主线程的 PCB 地址是 0xc009e000，对，确实如此。

正是因为 PCB 要占用一个自然页，所以，在低端 1MB 的内存布局中，明明有一段 0x7E00～0x9FBFF 的可用内存空间，我们偏偏把 0x9f000 作为内核栈顶（0xc009f000 对应的物理地址是 0x9f000，因为在页表中，我们把低端 1MB 的内存做的是对等映射），而不是 mov esp，0xc009fc00，还为浪费了 0xc00 字节的空间而疑惑。现在您是否有豁然开朗的感觉了？

不过话又说回来了，在我们的实现机制中，只要保证 PCB 的起始地址是自然页就好，因为程序在运行过程中，栈指针也会在 PCB 中不断上下游动，因此，在不破坏程序或线程信息的情况下，栈顶可以在此页内的任意部分。所以在原理上，用 mov esp，0xc009fc00 把主线程的栈顶初始化为 0xc009fc00 也是可以的，只是这样一来，主线程 main 的 PCB 就不完美了，别人的 PCB 都是 4096 字节，主线程的 PCB 少了 0x400 字节（0xc009fc00～0xc009ffff）。

我们的内核预计在 70KB 左右，装载到 0x9f000 以下是绰绰有余的，所以，主线程的栈顶地址 0x9f000 是我们在低端 1MB 中所用到的最高地址。

当前虚拟机配置了 32MB 的物理内存，这 32MB 物理内存需要 1024 字节的位图，也就是仅占四分之一页，故一页大小的位图可管理 128MB 的内存。对于目前的 32MB 内存来说，用一页内存来存储位图还浪费呢，不过为了扩展，我们可以更任性一点，在此打算支持 4 页内存的位图，即最大可管理 512MB 的物理内存。

既然 0xc009e000 已经是主线程的 PCB，一页大小为 0x1000，故再减去 4 页，即 0xc009e000 − 0x4000 = 0xc009a000。故我们的位图地址为 0xc009a000。

不知道大伙儿有没有想过，为什么一定要在低端 1MB 内存以内为位图选地址呢？暂时先留点伏笔，一会儿在介绍内存池初始化函数 mem_pool_init 时再揭晓答案。

第 15 行定义了宏 K_HEAP_START，用来表示内核所使用的堆空间起始虚拟地址，其值为 0xc0100000。

内核也是程序，它偶尔也需要动态申请内存来完成某项工作，动态申请的内存都是在堆空间中完成的，我们为此也定义了内核所用的堆空间，堆也是内存，内存就得有地址，从哪个地址开始分配呢？这就是 K_HEAP_START 的作用，堆的起始虚拟地址。

在 loader 中我们已经通过设置页表把虚拟地址 0xc0000000～0xc00fffff 映射到了物理地址 0x00000000～0x000fffff（低端 1MB 的内存），故我们为了让虚拟地址连续，将堆的起始虚拟地址设为 0xc0100000。当然，虚拟地址也可以不连续，不过让其连续不是显得紧凑一些吗？这并不是强制的。

这里给大伙提醒一下，物理地址 0x100000～0x101fff，是我们已经在 loader.S 中定义好的页目录及页表，因此将来的内核虚拟地址 0xc0100000～0xc0101fff 并不映射到这两个物理地址，必须要绕过它们。

第 17 行是内存管理的主角，物理内存池结构体，struct pool，用它来管理本内存池中的所有物理内存。此结构中定义了三个成员，pool_bitmap 是本内存池用于管理内存的位图结构，phy_addr_start 是本内存池的物理内存起始地址。pool_size 是本物理内存池的内存容量，因为物理地址是有限的。

您看，struct pool 与 struct virtual_addr 同样都是地址池结构，但和 struct pool 相比，struct virtual_addr 中没有 pool_size 成员，尽管虚拟地址空间最大是 4GB，但相对来说是无限的，不需要指定地址空间大小。因此虚拟地址池和物理地址池分别定义了两个结构。

第 24～25 行定义了内核物理内存池变量 kernel_pool 和用户物理内存池变量 user_pool，它们都是全局变量，以后的内存管理都需要用到这两个变量，它们在函数 mem_pool_init 中被初始化。

在继续介绍函数 mem_pool_init 之前，暂且移步代码 8-11-3 的最后几行，这里有函数 mem_pool_init 的调用，咱们先看看传进来的是什么参数。

在代码 8-11-3 中的第 94 行定义了变量 mem_bytes_total，它用来存储机器上安装的物理内存总量。大伙儿还记得内存容量是如何获取的吗？在 loader.S 中，为了获取内存容量，我们用了三种 BIOS 方法，最终把获取到的内存容量保存在汇编变量 total_mem_bytes 中，其物理地址为 0xb00。total_mem_bytes 是用伪指令 dd 来定义的，其宽度是 32 位，因此，我们先把 0xb00 转换成 32 位整型指针，再通过*对该指针做取值操作，这样就获取到了内存容量。这就是代码“*(*(uint32_t*)(0xb00))”的意义。

接下来在第 95 行将此内存容量值传给函数 mem_pool_init，让函数 mem_pool_init 将它分配给各物理内存池，好，咱们再回来细说代码 8-11-1 中 mem_pool_init 函数的实现。

函数 mem_pool_init 接受一个参数，all_mem，此参数表示内存容量，函数的功能是根据内存容量 all_mem 的大小初始化物理内存池的相关结构。

是时候揭开之前留下的伏笔了，为什么要将位图地址选在低端 1MB 以下呢？

内存管理中所使用的数据结构必然也要保存在内存中，即内存管理系统自己也要占用内存，那它还要管理自己所占的内存吗？自己管理自己似乎有点“复杂”，不要再动脑筋了，其实用不着那么麻烦的，一般的内存管理系统所管理的是那些空闲的内存，即已被使用的内存是不在内存池中的，“已使用的内存”当然包括内存管理相关数据结构所占的内存，位图就是用于管理内存的数据结构，这也是位图地址选为 0xc009a000（0x9a000）的原因，此地址位于低端 1MB 之内，这里面的内存几乎都被占用了，因此我们就不用考虑它占用的内存了。下面咱们看看具体的实现吧。

第 30 行定义了变量 page_table_size，它用来记录页目录表和页表占用的字节大小，总大小等于页目录表大小+页表大小。页目录大小为 1 页框，第 0 和第 768 个页目录项指向同一个页表，它们共享这 1 页框空间，第 769～1022 个页目录项共指向 254 个页表，故页表总大小等于 256*PG_SIZE，共计 0x200000 字节，2MB。注意，最后一个页目录项（第 1023 个 pde）指向页目录表，因此不重复计算空间。

第 32 行定义了变量 used_mem，它用来记录当前已使用的内存字节数，这包括页表大小 page_table_size 和低端 0x100000 字节（1MB）内存。

第 33 行定义变量 free_mem，它用来存储目前可用的内存字节数，用总内存 all_mem 减去 used_mem 便是 free_mem。

第 34 行定义了变量 all_free_pages，它用来保存可用内存字节数 free_mem 转换成的物理页数，因为内存池中的内存单位是物理页。

此函数的其余部分还要看代码 8-11-2 和代码 8-11-3。咱们继续。

代码 8-11-2　（project/c8/d/kernel/memory.c）

```
36      uint16_t kernel_free_pages = all_free_pages / 2;
37      uint16_t user_free_pages = all_free_pages - kernel_free_pages;
38
39 /* 为简化位图操作，余数不处理，坏处是这样做会丢内存。
40 好处是不用做内存的越界检查，因为位图表示的内存少于实际物理内存*/
41      uint32_t kbm_length = kernel_free_pages / 8;
// Kernel BitMap 的长度，位图中的一位表示一页，以字节为单位
42      uint32_t ubm_length = user_free_pages / 8;
// User BitMap 的长度
43
44      uint32_t kp_start = used_mem;
// Kernel Pool start，内核内存池的起始地址
45      uint32_t up_start = kp_start + kernel_free_pages * PG_SIZE;
// User Pool start，用户内存池的起始地址
46
47      kernel_pool.phy_addr_start = kp_start;
48      user_pool.phy_addr_start   = up_start;
49
50      kernel_pool.pool_size = kernel_free_pages * PG_SIZE;
51      user_pool.pool_size   = user_free_pages * PG_SIZE;
52
53      kernel_pool.pool_bitmap.btmp_bytes_len = kbm_length;
54      user_pool.pool_bitmap.btmp_bytes_len   = ubm_length;
55
56 /*********    内核内存池和用户内存池位图   ***********
57  *   位图是全局的数据，长度不固定。
58  *   全局或静态的数组需要在编译时知道其长度，
59  *   而我们需要根据总内存大小算出需要多少字节，
60  *   所以改为指定一块内存来生成位图。
61  *   ************************************************/
62 // 内核使用的最高地址是 0xc009f000，这是主线程的栈地址
   //（内核的大小预计为 70KB 左右）
63 // 32MB 内存占用的位图是 2KB
   //内核内存池的位图先定在 MEM_BITMAP_BASE(0xc009a000)处
64    kernel_pool.pool_bitmap.bits = (void*)MEM_BITMAP_BASE;
65
66 /* 用户内存池的位图紧跟在内核内存池位图之后 */
67    user_pool.pool_bitmap.bits = (void*)(MEM_BITMAP_BASE + kbm_length);
```

第 36～37 行定义的变量 kernel_free_pages 用来存储分配给内核的空闲物理页，变量 user_free_pages 把分配给内核后剩余的空闲物理页作为用户内存池的空闲物理页数量。

第 41 行定义了变量 kbm_length，它用来记录位图的长度。因为位图中的 1 位表示 1 页，故用 kernel_free_pages 除以 8 以获得位图的长度。为方便写程序，余数咱们就不处理了，这样做的好处是不用做内存的越界检查，因为位图表示的内存少于实际物理内存，坏处是会丢 (1～7 页)*2 的内存：内核物理内存池+用户物理内存池。

第 42 行的 ubm_length 是用户内存池位图的长度，同 kbm_length 一个道理。

第 44 行的 kp_start 用于记录内核物理内存池的起始地址，值就是 used_mem。

第 45 行的 up_start 用于记录用户物理内存池的起始地址，其值等于 kp_start 加上内核物理内存池中的内存字节数，即加上 kernel_free_pages * PG_SIZE。

第 47～48 行用以上的两个起始物理地址初始化各自内存池的起始地址，即 kernel_pool.phy_ addr_start = kp_start，user_pool.phy_addr_start　= up_start。

第 50～51 行用各自内存池中的容量字节数（物理页数乘以 PG_SIZE）初始化各自内存池的 pool_size。

第 53～54 行用各自内存池的位图长度 kbm_length 和 ubm_length 初始化各自内存池的位图中的位图字节长度成员 btmp_bytes_len。

第 56～67 行是初始化各自内存池所使用的位图。

这里有多行注释，解释了为什么要用此方式定义位图。

在上一节和大伙介绍位图的时候已经提到过，位图可以用数组来实现，但是位图是全局数据结构，全

局或静态的数组需要在编译时知道其长度，而位图的长度取决于具体要管理的内存页数量，因此是无法预计的，所以我们就只在 struct bitmap 结构中定义了 bits 成员用来记录上层模块提供的位图的地址。这里提到的上层模块，就是指此处的 mem_pool_init 函数，由此函数为 struct bitmap 中的 bits 提供位图地址。

也许您会说，既然位图的长度并不固定，是否可以用变长数组来实现？变长数组只在 c99 中支持，并且数组占用的内存是堆空间，而且那还是在操作系统的支持下，而我们目前所做的正是在构建操作系统，而本节我们的内存管理系统，其实也是在构建堆内存管理，我们的两个内存池就相当于堆。

综上所述，由于编译器必须要“事先”知道数组长度才能定义静态数组，但我们用作位图的数组，其长度取决于内存大小，我们需要在程序运行过程中根据总内存大小算出位图数组需要多少字节，也就是说在“将来”才能确定数组长度。因此改为指定一块内存来生成位图，这样就不需要固定长度了。

第 64 行初始化内核位图的指针 kernel_pool.pool_bitmap.bits，其值为 MEM_BITMAP_BASE，也就是 0x9a000。

第 67 行初始化用户位图的指针 user_pool.pool_bitmap.bits，其值为 MEM_BITMAP_BASE + kbm_length，也就是紧跟在内核位图之后。

还有一些初始化工作在代码 8-11-3 中，大伙儿继续跟我来。

代码 8-11-3　（project/c8/d/kernel/memory.c）

```
68 /********************输出内存池信息**********************/
69 put_str("     kernel_pool_bitmap_start:");
   put_int((int)kernel_pool.pool_bitmap.bits);
70 put_str(" kernel_pool_phy_addr_start:");
   put_int(kernel_pool.phy_addr_start);
71      put_str("\n");
72      put_str("user_pool_bitmap_start:");
        put_int((int)user_pool.pool_bitmap.bits);
73      put_str(" user_pool_phy_addr_start:");
        put_int(user_pool.phy_addr_start);
74      put_str("\n");
75
76      /* 将位图置 0*/
77      bitmap_init(&kernel_pool.pool_bitmap);
78      bitmap_init(&user_pool.pool_bitmap);
79
80      /* 下面初始化内核虚拟地址的位图，按实际物理内存大小生成数组。*/
81      kernel_vaddr.vaddr_bitmap.btmp_bytes_len = kbm_length;
        // 用于维护内核堆的虚拟地址，所以要和内核内存池大小一致
82
83      /* 位图的数组指向一块未使用的内存，
        目前定位在内核内存池和用户内存池之外*/
84      kernel_vaddr.vaddr_bitmap.bits = \
        (void*)(MEM_BITMAP_BASE + kbm_length + ubm_length);
85
86      kernel_vaddr.vaddr_start = K_HEAP_START;
87      bitmap_init(&kernel_vaddr.vaddr_bitmap);
88      put_str("   mem_pool_init done\n");
89 }
90
91 /* 内存管理部分初始化入口 */
92 void mem_init() {
93     put_str("mem_init start\n");
94     uint32_t mem_bytes_total = (*(uint32_t*)(0xb00));
95     mem_pool_init(mem_bytes_total);       // 初始化内存池
96     put_str("mem_init done\n");
97 }
```

第 69～74 行是打印内存池的信息，这包括内存池的所用位图的起始地址和内存池的起始物理地址。

内存池中的位图还需要初始化，位值为 0 表示该位对应的内存页未分配，位值为 1 表示该位对应的内存页已分配。因此在第 77～78 行调用函数 bitmap_init 将位图初始化为 0。

第 80 行之后开始初始化内核虚拟地址池，方法同前面所述类似，在第 84 行为其所使用的位图指针初始化，将其安排在紧挨着内核内存池和用户内存池所用的位图之后，即：

```
kernel_ vaddr.vaddr_bitmap.bits=(void*)(MEM_BITMAP_BASE + kbm_length + ubm_length)
```

虚拟内存池的起始地址为 K_HEAP_START，即 0xc0100000。之后在第 87 行调用 bitmap_init 将其位图初始化。至此，mem_pool_init 函数到这就说完了。

在第 92 行的是函数 mem_init，mem_pool_init 是在此函数中调用的，因此 mem_init 是内存管理的初始化入口，从名字上就能看出，它和之前的 idt_init、timer_init 一样，要被放到 init.c 中的 init_all 函数中才行，这里就不再演示了。

在 makefile 中添加 memory.o 的编译及链接信息后，执行 make all，最新的 kernel.bin 便被写入 bochs 虚拟硬盘了（好方便）。在 bochs 上运行的结果如图 8-20 所示。

```
I am kernel
init_all
idt_init start
   idt_desc_init done
   pic_init done
idt_init done
timer_init start
timer_init done
mem_init start
   mem_pool_init start
      kernel_pool_bitmap_start:C009A000 kernel_pool_phy_addr_start:200000
      user_pool_bitmap_start:C009A1E0 user_pool_phy_addr_start:1100000
   mem_pool_init done
mem_init done
CTRL + 3rd button enables mouse  NUM  CAPS  SCRL  HD:0-M  HD:0-S
```

▲图 8-20　内存池信息

您看，我们打印出了内存池的信息。其中内核物理内存池所用的位图地址在 0xc009a000，其内存池中第一块物理页地址是 0x200000。下一行的是对应的用户物理内存池的相关信息。以后在申请内存时，都会更改位图的内容，我们可以通过查看位图地址处的值来判断内存系统工作是否正确。

本节到此结束，下一节中咱们试着分配内存。

8.5.2　内存管理系统第一步，分配页内存

在有了内存池之后，咱们可以采取进一步的行动了。咱们在 C 语言下是用 malloc 函数向操作系统申请内存的，此函数可以申请的内存数量比较灵活，在用户眼里，可以申请任意字节尺寸的内存。其实它后面的故事可多呢，先留点悬念，以后咱们实现 malloc 时就知道了。本节要做的是实现任意内存分配的基础部分，“整页分配”。

“整页分配”这个词是我自己杜撰的，我的意思是咱们先支持一次分配 n 个页的内存，即 n*4096 字节，看上去有点粗狂，其实用途可大呢。

在上一节的基础上，咱们又对 memory.h 和 memory.c 做出了改进，memory.h 见代码 8-12-1，memory.c 的最新面貌见代码 8-12-2 和代码 8-12-3，咱们先从头文件开始，里面有新定义的结构。

代码 8-12-1　（project/c8/e/kernel/memory.h）

```
 6 /* 内存池标记，用于判断用哪个内存池 */
 7 enum pool_flags {
 8     PF_KERNEL = 1,    // 内核内存池
 9     PF_USER = 2       // 用户内存池
10 };
11
12 #define  PG_P_1   1     // 页表项或页目录项存在属性位
13 #define  PG_P_0   0     // 页表项或页目录项存在属性位
14 #define  PG_RW_R  0     // R/W 属性位值，读/执行
15 #define  PG_RW_W  2    // R/W 属性位值，读/写/执行
16 #define  PG_US_S  0     // U/S 属性位值，系统级
17 #define  PG_US_U  4     // U/S 属性位值，用户级
```

我们在 C 语言中申请内存时，由于我们是用户程序，操作系统直接会在用户内存池中分配内存，但这对应到内核中具体的操作时，必须要“显式”指定在哪个内存池中申请，故我们在 memory.h 中新增了枚举结构 enum pool_flags 用来区分这两个内存池，此结构里面定义了两个成员，PF_KERNEL 值为 1，它

代表内核物理内存池。PF_USER 值为 2，它代表用户物理内存池。

内存管理中，必不可少的操作就是修改页表，这势必涉及到页表项及页目录项的操作，因此又在 memory.h 中定义了一些 PG_开头的宏，这是页表项或页目录项的属性，memory.c 中的函数会用到它们。

简要介绍下这些属性值，以下所说的页内存表示页表或普通物理页。

PG 前缀表示页表项或页目录项，US 表示第 2 位的 US 位，RW 表示第 1 位的 RW 位，P 表示第 0 位的 P 位。

- PG_P_1 表示 P 位的值为 1，表示此页内存已存在。
- PG_P_0 表示 P 位的值为 0，表示此页内存不存在。
- PG_RW_W 表示 RW 位的值为 W，即 RW=1，表示此页内存允许读、写、执行。
- PG_RW_R 表示 RW 位的值为 R，即 RW=0，表示此页内存允许读、执行。
- PG_US_S 表示 US 位的值为 S，即 US=0，表示只允许特权级别为 0、1、2 的程序访问此页内存，3 特权级程序不被允许。
- PG_US_U 表示 US 位的值为 U，即 US=1，表示允许所有特权级别程序访问此页内存。

以上各属性的值是以它们的位次来定义的，并不是 0 或 1，这样方便后面的页表项或页目录项的属性合成。

头文件就更新了这些内容，在介绍 memory.c 之前，咱们先来点干货。为减少学习阻力，先给大伙复习下 32 位虚拟地址的转换过程。

（1）高 10 位是页目录项 pde 的索引，用于在页目录表中定位 pde，细节是处理器获取高 10 位后自动将其乘以 4，再加上页目录表的物理地址，这样便得到了 pde 索引对应的 pde 所在的物理地址，然后自动在该物理地址中，即该 pde 中，获取保存的页表物理地址（为了严谨，说的都有点拗口了）。

（2）中间 10 位是页表项 pte 的索引，用于在页表中定位 pte。细节是处理器获取中间 10 位后自动将其乘以 4，再加上第一步中得到的页表的物理地址，这样便得到了 pte 索引对应的 pte 所在的物理地址，然后自动在该物理地址（该 pte）中获取保存的普通物理页的物理地址。

（3）低 12 位是物理页内的偏移量，页大小是 4KB，12 位可寻址的范围正好是 4KB，因此处理器便直接把低 12 位作为第二步中获取的物理页的偏移量，无需乘以 4。用物理页的物理地址加上这低 12 位的和便是这 32 位虚拟地址最终落向的物理地址。

32 位地址经过以上三步拆分，地址最终落在某个物理页内。

注意啦，再提醒一次，页表的作用是将虚拟地址转换成物理地址，此工作表面虚幻，但内心真实，其转换过程中涉及访问的页目录表、页目录项及页表项，都是通过真实物理地址访问的，否则若用虚拟地址访问它们的话，会陷入转换的死循环中不可自拔。

以上我一直强调的是物理地址，此概念在后面的函数中会屡屡出现，希望大伙儿能够体会到我的良苦用心。

回忆至此结束，咱们继续开工啦。

代码 8-12-2 （project/c8/e/kernel/memory.c）

```
 1 #include "memory.h"
 2 #include "bitmap.h"
 3 #include "stdint.h"
 4 #include "global.h"
 5 #include "debug.h"
 6 #include "print.h"
 7 #include "string.h"
…略
18 #define PDE_IDX(addr) ((addr & 0xffc00000) >> 22)
19 #define PTE_IDX(addr) ((addr & 0x003ff000) >> 12)
…略
34 /* 在pf表示的虚拟内存池中申请pg_cnt个虚拟页,
35  * 成功则返回虚拟页的起始地址，失败则返回NULL */
36 static void* vaddr_get(enum pool_flags pf, uint32_t pg_cnt) {
37     int vaddr_start = 0, bit_idx_start = -1;
38     uint32_t cnt = 0;
39     if (pf == PF_KERNEL) {
40         bit_idx_start  = bitmap_scan(&kernel_vaddr.vaddr_bitmap, pg_cnt);
41         if (bit_idx_start == -1) {
42             return NULL;
```

```
43         }
44         while(cnt < pg_cnt) {
45             bitmap_set(&kernel_vaddr.vaddr_bitmap, bit_idx_start + cnt++, 1);
46         }
47         vaddr_start = kernel_vaddr.vaddr_start + bit_idx_start * PG_SIZE;
48     } else {
49     // 用户内存池，将来实现用户进程再补充
50     }
51     return (void*)vaddr_start;
52 }
53
54 /* 得到虚拟地址 vaddr 对应的 pte 指针*/
55 uint32_t* pte_ptr(uint32_t vaddr) {
56      /* 先访问到页表自己 + \
57       * 再用页目录项 pde（页目录内页表的索引）作为 pte 的索引访问到页表 + \
58       * 再用 pte 的索引作为页内偏移*/
59     uint32_t* pte = (uint32_t*)(0xffc00000 + \
60           ((vaddr & 0xffc00000) >> 10) + \
61           PTE_IDX(vaddr) * 4);
62     return pte;
63 }
64
65 /* 得到虚拟地址 vaddr 对应的 pde 的指针 */
66 uint32_t* pde_ptr(uint32_t vaddr) {
67     /* 0xfffff 用来访问到页表本身所在的地址 */
68     uint32_t* pde = (uint32_t*)((0xfffff000) + PDE_IDX(vaddr) * 4);
69     return pde;
70 }
71
72 /* 在 m_pool 指向的物理内存池中分配 1 个物理页，
73  * 成功则返回页框的物理地址，失败则返回 NULL */
74 static void* palloc(struct pool* m_pool) {
75      /* 扫描或设置位图要保证原子操作 */
76      int bit_idx = bitmap_scan(&m_pool->pool_bitmap, 1);  // 找一个物理页面
77      if (bit_idx == -1 ) {
78          return NULL;
79      }
80      bitmap_set(&m_pool->pool_bitmap, bit_idx, 1);    // 将此位 bit_idx 置 1
81      uint32_t page_phyaddr = ((bit_idx * PG_SIZE) + m_pool->phy_addr_start);
82      return (void*)page_phyaddr;
83 }
84
```

代码 8-12-2 的前几行新增了一些头文件。

第 18～19 行定义了两个宏，PDE_IDX 用于返回虚拟地址的高 10 位，即 pde 索引部分，此部分用于在页目录表中定位 pde。

PTE_IDX 用于返回虚拟地址的中间 10 位，即 pte 索引部分，此部分用于在页表中定位 pte。

vaddr_get 函数接受两个参数，pf 是内存池的 flag，就是在头文件中定义的 enum pool_flags。pg_cnt 是页数，函数的功能是在 pf 表示的虚拟内存池中申请 pg_cnt 个虚拟页，若申请成功，则返回虚拟页的起始地址，失败时，则返回 NULL。

函数中定义的变量 vaddr_start 用于存储分配的起始虚拟地址，bit_idx_start 用于存储位图扫描函数 bitmap_scan 的返回值，默认为−1。

接下来会判断 pf 的值，如果其等于 PF_KERNEL，便认为是在内核虚拟地址池中申请地址，于是调用 bitmap_scan 函数扫描内核虚拟地址池中的位图。若 bitmap_scan 返回−1，则 vaddr_get 函数返回 NULL。

由于目前只是试探着扫描了位图，并未将位图中的相应位置 1，所以在后面用 while 循环，根据申请的页数量，即 pg_cnt 的值，逐次调用 bitmap_set 函数将相应位置 1。

将位图置 1 之后，工作基本上完成了，现在需要将 bit_idx_start 转换为虚拟地址，如代码“vaddr_start = kernel_vaddr.vaddr_start + bit_idx_start * PG_SIZE”，因为位图中的一位代表实际 1 页大小的内存，所以转换原理还是很简单的，就是用虚拟内存池的起始地址 kernel_vaddr. vaddr_start 加上起始位索引 bit_idx_start 相对于内存池的虚拟页偏移地址 bit_idx_start * PG_SIZE。

下面的 else 对应的部分是用户内存池，由于目前尚未实现用户进程，先空着吧。

最后用“return (void*)vaddr_start”将 vaddr_star 转换成指针后返回。

第 55 行定义了函数 pte_ptr，它接受一个参数 vaddr，功能是得到地址 vaddr 所在 pte 的指针，强调下，指针的值也就是虚拟地址，故此函数实际返回的是能够访问 vaddr 所在 pte 的虚拟地址。

此函数涉及到页表操作，现在大伙儿回忆一下，之前咱们在 loader.S 中创建页表的时候，在最后一个页目录项里写的是页目录表自己的物理地址，目的就是为了通过此页目录项编辑页表，如今终于派上用场了（耳边突然响起了:“可喜可贺……”玩笑玩笑）。

由于此函数的目的是获取地址 vaddr 所在页表项 pte 的地址，这说明咱们要访问的内存是页表，是不是觉得有些神秘呢？

访问页表也得需要用内存地址来访问，在分页机制下任何地址都是虚拟地址，因此我们要根据 vaddr 构造出一新的虚拟地址，暂且称之为 new_vaddr，用它来访问 vaddr 本身所在的 pte 的物理地址。对，您没有看错，虚拟地址最终会落到某个物理地址上，咱们就是要用虚拟地址访问某个确切的物理地址。

理下思路，处理器处理 32 位地址的三个步骤如下。

（1）首先处理高 10 位的 pde 索引，从而处理器得到页表物理地址。

（2）其次处理中间 10 位的 pte 索引，进而处理器得到普通物理页的物理地址。

（3）最后是把低 12 位作为普通物理页的页内偏移地址，此偏移地址加上物理页的物理地址，得到的地址之和便是最终的物理地址，处理器到此物理地址上进行读写操作。

也就是说，我们要创造的这个新的虚拟地址 new_vaddr，它经过处理器以上三个步骤的拆分处理，最终会落到 vaddr 自身所在的 pte 的物理地址上。

pte 位于页表中，因此要想访问 vaddr 所在的 pte，必须保证处理器在第 2 步处理 pte 索引时得到的是页表的物理地址，而不是普通物理页的物理地址，这样可以再利用第 3 步中的低 12 位做页表内的偏移量，用此偏移量加上页表物理地址，所得的地址之和便是 vaddr 所在的 pte 的物理地址。

简单来说就是要想获取 pte 的地址，必须先访问到页目录表，再通过其中的页目录项 pde，找到相应的页表，在页表中才是页表项 pte。因此，我们需要分别在地址的高 10 位、中间 10 位和低 12 位中填入合适的数，拼凑出满足此要求的新的 32 位地址 new_vaddr。

拼凑新地址 new_vaddr 的过程可分为三步。

- 第一步，先访问到页目录表。

首先，处理器需要高 10 位来定位 pde。

32 位地址中，高 10 位用于定位页目录项，由于最后一个页目录项保存的正是页目录表物理地址，按照这个思路，我们需要让地址的高 10 位指向最后一个页目录项，即第 1023 个 pde，这样才能获取页目录表本身的物理地址。

1023 换算成十六进制是 0x3ff，将其移到高 10 后，变成 0xffc00000。于是，0xffc00000 让处理器自动在最后一个 pde 中取出页目录表物理地址，此处页目录表物理地址为 0x100000，如果忘记的话，可以看看 boot/include/boot.inc 中的配置项 PAGE_DIR_TABLE_POS，其值便为 0x100000。

在咱们眼里，最后一个 pde 中的物理地址是页目录表地址，因为这是咱们在创建页表时提前安排好的，但处理器把保存在 pde 中的地址都视为页表地址，即处理器会把刚刚获得的页目录表当成页表来处理，这是我们第 1 次骗了处理器。

此时我们拼凑出了新虚拟地址 new_vaddr 的高 10 位，下面继续努力。

- 第二步，找到页表。

其次，处理器需要 pte 索引。

为满足处理器的要求，我们需要再凑出中间 10 位。中间 10 位是页表项的索引，用来在页表中定位页表项 pte。我们在上一步中已经得到了页目录表物理地址（其实处理器把页目录表当成页表了），页表地址保存在页目录项中，因此我们要先想办法访问到 vaddr 所在的页目录项。此时处理器已经把上一步获得的页目录表当成了页表，其需要的是 pte 的索引，因此我们把 vaddr 的 pde 索引当作处理器视角中的 pte 索引就行了，现在要做的是将参数 vaddr 的高 10 位（pde 索引）取出来，做新地址 new_vaddr 的中间 10 位（pte 索引）。

于是我们先用按位与操作“(vaddr & 0xffc00000)”获取高 10 位，再将其右移 10 位，使其变成中间 10 位，于是就成了处理器眼中的 pte 索引。这样在处理器处理新地址 new_vaddr 的 pte 索引时，以为接下来获得的是 pte 中的普通物理页地址，但这只是处理器视角中的情景。而事实上是我们骗了处理器，由于上一步我们获得的是页目录表地址，并且本步中传给它的 pte 索引是 vaddr 中的 pde 索引，故此时处理器获得的是 vaddr 中高 10 位的 pde 索引所对应的 pde 里保存的页表的物理地址，并不是 pte 中保存的普通物理页的物理地址。这是我们第 2 次骗了处理器。所以，此时我们获得了 vaddr 所在的页表物理地址。

此时我们已拼凑出了新虚拟地址 new_vaddr 的中间 10 位。

- 第三步，在页表中找到 pte。

最后，处理器需要地址的低 12 位。

我们一次又一次地骗了处理器，马上还要再骗一次呢，不过话说回来了，人家处理器才不关心是否被骗，因为由欺骗导致的错误，受害者不是处理器，而是欺骗它的人，所以我们在欺骗处理器时要确保自己不是受害者，这是一个骗子最起码的“素质”，哈哈，玩笑。

上一步中处理器认为已经找到了最终的物理页地址，所以它此时需要的是 32 位地址中的低 12 位，用该 12 位作为上一步中获取到的物理页的偏移量，当然，这依然只是处理器的视角。在我们眼里，上一步获得的是页表的物理地址，因此我们只要把 vaddr 的中间 10 位转换成处理器眼里的 12 位长度的页内偏移量就行了。由于地址的低 12 位寻址范围正好是一页的 4KB 大小，故处理器直接拿低 12 位去寻址，不会再为其自动乘以 4，因此，咱们得手动将 vaddr 的 pte 部分乘 4 后再交给处理器。这里的做法是先用 PTE_IDX(vaddr) 获取 vaddr 的 pte 索引，即中间 10 位，再将其乘 4，即 PTE_IDX(vaddr) * 4 拼凑出了新虚拟地址 new_vaddr 的低 12 位。故 0xffc00000 + ((vaddr & 0xffc00000) >> 10) + PTE_IDX(vaddr) * 4 的结果就是最终的新虚拟地址 new_vaddr 的完整 32 位数值，new_vaddr 保存在指针变量 pte 中。由于此结果仅仅是个整型数值，需要将其通过强制类型转换，即(uint32_t*)，转换成 32 位整型地址。此时指针变量 pte 指向 vaddr 所在的 pte。最后通过 return pte 将此指针返回。

函数 pde_ptr 只接受一个参数，它的功能是得到虚拟地址 vaddr 所在 pde 的指针，也就是返回能够访问该 pde 的虚拟地址。用此指针可以访问到虚拟地址 vaddr 对应的 pde。此函数的实现原理和 pte_ptr 一样，不过更简单一些。

同样，访问 pde 也必须要通过地址，因此我们这里也必须要根据 vaddr 来构造一个新的 32 位地址 new_vaddr。根据前面所说的处理器处理 32 位地址的三个步骤，处理器先处理的是 32 位地址中高 10 位的 pde 索引，其次是中间 10 位的 pte 索引，最后是低 12 位。

我们要创造的这个新的虚拟地址 new_vaddr，它经过以上三个步骤的拆分处理，最终处理器会把要访问的地址落在 vaddr 所在的 pde 的物理地址上。

由于要访问的是 vaddr 所在的页目录项 pde，所以必须想办法在第 2 步中让处理器处理 pte 索引时获得的是页目录表物理地址，然后利用低 12 位作为物理页的偏移量，此偏移量加上页目录表的物理地址，所得的地址之和便是 vaddr 所在的 pde 的物理地址。

其实早在 long long ago 介绍页表的时候就和大伙儿说过了，由于最后一个页目录项中存储的是页目录表物理地址，故当 32 位地址中高 20 位为 0xfffff 时，这就表示访问到的是最后一个页目录项，即获得了页目录表物理地址。这也很容易理解，0xfffffxxx 的高 10 位是 0x3ff，中间 10 位也是 0x3ff，也就是处理 pde 索引时得到的是页目录表的物理地址，此时处理器以为此页目录表就是页表，继续用 pte 索引在该页表（页目录表）找到最后一个页表项 pte（其实是页目录项 pde），所以再次获得了页目录表物理地址（当然处理器以为获得的是普通物理页的物理地址）。

因此，新虚拟地址 new_vaddr 等于 0xfffff000 再加上 vaddr 的页目录项索引乘以 4 的积，即“(0xfffff000) + PDE_IDX(vaddr) * 4”。此时的 new_vaddr 便落到 vaddr 所在的页目录项 pde 的物理地址上。由于此结果仅仅是个整型数值，需要将其通过强制类型转换成 32 位整型指针。最终的新虚拟地址 new_vaddr 保存在指针变量 pde 中，因此“pde = (uint32_t*)((0xfffff000) + PDE_IDX(vaddr) * 4)”，此时指针变量 pde 指向了 vaddr 所在的 pde，最后通过 return pde 将指针返回。

两件事要注意。

（1）pte_ptr 和 pde_ptr 这两个函数返回的是能够访问到 vaddr 所在 pte 及 pde 的新虚拟地址 new_vaddr，new_vaddr 经过处理器处理 32 位地址的三个步骤，最终指向 vaddr 的 pte 及 pde 所在的物理地址。因此，这两个函数的功能等同于：给我一个新的虚拟地址 new_vaddr，让它指向 vaddr 所在的 pde 及 pte，也就是让 new_vaddr 指向 pde 及 pte 所在的物理地址。

（2）这两个函数中的参数 vaddr，可以是已经分配、在页表中存在的，也可以是尚未分配，目前页表中不存在的虚拟地址，pte_ptr 和 pde_ptr 这两个函数只是根据虚拟地址转换的规则计算出 vaddr 对应的 pte 及 pde 的虚拟地址，与 vaddr 所在的 pte 及 pde 是否存在无关。

这两个函数是修改页表的核心，相对来说有点难，不过大伙儿请放心，不会再有比它们更难的函数了。

palloc 函数只接受一个参数 m_pool，功能是在 m_pool 指向的物理内存池中分配 1 个物理页，成功时则返回页框的物理地址，失败时则返回 NULL。

定义的变量 bit_idx 用于存储 bitmap_scan 函数的返回值，bitmap_scan 函数在物理内存池的位图中查找可用位，如果失败，则返回−1，因此函数 palloc 也将返回 NULL，宣告失败。若 bitmap_scan 的返回值不为−1，也就是找到了可用位，接下来再通过函数 bitmap_set 将 bit_idx 位设置为 1，也就是代码“bitmap_set(&m_pool->pool_bitmap, bit_idx, 1)”。

变量 page_phyaddr 用于保存分配的物理页地址，它的值是物理内存池的起始地址 m_pool-> phy_addr_start+ 物理页在内存池中的偏移地址（bit_idx * PG_SIZE）。

最后通过“return (void*) page_phyaddr”将物理页地址转换成空指针后返回。

下面继续看代码 8-12-3。

代码 8-12-3　（project/c8/e/kernel/memory.c）

```
 85 /* 页表中添加虚拟地址_vaddr 与物理地址_page_phyaddr 的映射 */
 86 static void page_table_add(void* _vaddr, void* _page_phyaddr) {
 87      uint32_t vaddr = (uint32_t)_vaddr, page_phyaddr = (uint32_t)_page_phyaddr;
 88      uint32_t* pde = pde_ptr(vaddr);
 89      uint32_t* pte = pte_ptr(vaddr);
 90
 91 /************************   注意   ************************
 92  * 执行*pte，会访问到空的 pde。所以确保 pde 创建完成后才能执行*pte，
 93  * 否则会引发 page_fault。因此在*pde 为 0 时，
    *pte 只能出现在下面 else 语句块中的*pde 后面。
 94  **********************************************************/
 95      /* 先在页目录内判断目录项的 P 位，若为 1，则表示该表已存在 */
 96      if (*pde & 0x00000001) {
   // 页目录项和页表项的第 0 位为 P，此处判断目录项是否存在
 97           ASSERT(!(*pte & 0x00000001));
 98
 99           if (!(*pte & 0x00000001)) {
   // 只要是创建页表，pte 就应该不存在，多判断一下放心
100                *pte = (page_phyaddr | PG_US_U | PG_RW_W | PG_P_1);
   // US=1,RW=1,P=1
101           } else {//目前应该不会执行到这，因为上面的 ASSERT 会先执行
102                PANIC("pte repeat");
103                *pte = (page_phyaddr | PG_US_U | PG_RW_W | PG_P_1);
   // US=1,RW=1,P=1
104           }
105      } else {    // 页目录项不存在，所以要先创建页目录再创建页表项
106           /* 页表中用到的页框一律从内核空间分配 */
107           uint32_t pde_phyaddr = (uint32_t)palloc(&kernel_pool);
108
109           *pde = (pde_phyaddr | PG_US_U | PG_RW_W | PG_P_1);
110
111           /* 分配到的物理页地址 pde_phyaddr 对应的物理内存清 0，
112            * 避免里面的陈旧数据变成了页表项，从而让页表混乱。
113            * 访问到 pde 对应的物理地址，用 pte 取高 20 位便可。
114            * 因为 pte 基于该 pde 对应的物理地址内再寻址，
115            * 把低 12 位置 0 便是该 pde 对应的物理页的起始*/
116           memset((void*)((int)pte & 0xfffff000), 0, PG_SIZE);
117
118           ASSERT(!(*pte & 0x00000001));
```

```
119             *pte = (page_phyaddr | PG_US_U | PG_RW_W | PG_P_1);
      // US=1,RW=1,P=1
120        }
121 }
122 
123 /* 分配 pg_cnt 个页空间，成功则返回起始虚拟地址，失败时返回 NULL */
124 void* malloc_page(enum pool_flags pf, uint32_t pg_cnt) {
125     ASSERT(pg_cnt > 0 && pg_cnt < 3840);
126 /***********   malloc_page 的原理是三个动作的合成：   ***********
127    1 通过 vaddr_get 在虚拟内存池中申请虚拟地址
128    2 通过 palloc 在物理内存池中申请物理页
129    3 通过 page_table_add 将以上得到的虚拟地址和物理地址在页表中完成映射
130 ***************************************************************/
131    void* vaddr_start = vaddr_get(pf, pg_cnt);
132    if (vaddr_start == NULL) {
133        return NULL;
134    }
135 
136    uint32_t vaddr = (uint32_t)vaddr_start, cnt = pg_cnt;
137    struct pool* mem_pool = pf & PF_KERNEL ? &kernel_pool : &user_pool;
138 
139    /* 因为虚拟地址是连续的，但物理地址可以是不连续的，所以逐个做映射*/
140    while (cnt-- > 0) {
141       void* page_phyaddr = palloc(mem_pool);
142       if (page_phyaddr == NULL) {  //失败时要将曾经已申请的虚拟地址和
                                  //物理页全部回滚，在将来完成内存回收时再补充
143          return NULL;
144       }
145       page_table_add((void*)vaddr, page_phyaddr); // 在页表中做映射
146       vaddr += PG_SIZE;         // 下一个虚拟页
147    }
148    return vaddr_start;
149 }
150 
151 /* 从内核物理内存池中申请 1 页内存，
    成功则返回其虚拟地址，失败则返回 NULL */
152 void* get_kernel_pages(uint32_t pg_cnt) {
153    void* vaddr =  malloc_page(PF_KERNEL, pg_cnt);
154    if (vaddr != NULL) {    // 若分配的地址不为空，将页框清 0 后返回
155        memset(vaddr, 0, pg_cnt * PG_SIZE);
156    }
157    return vaddr;
158 }
```

第 86 行定义了函数 page_table_add，它接受两个参数，虚拟地址_vaddr 和物理地址_page_phyaddr，功能是添加虚拟地址_vaddr 与物理地址_page_phyaddr 的映射。

虚拟地址和物理地址的映射关系是在页表中完成的，本质上是在页表中添加此虚拟地址对应的页表项 pte，并把物理页的物理地址写入此页表项 pte 中。也就是说，页表操作会用到上面介绍过的 pde_ptr 和 pte_ptr 函数。

函数开头调用了函数 pde_ptr(vaddr)，以此获取虚拟地址 vaddr 所在的 pde 的虚拟地址，此地址用指针变量 pde 保存。

接着调用 pte_ptr(vaddr)获得 vaddr 所在的 pte 虚拟地址，此地址保存在指针变量 pte 中。

pte 隶属于某个页表，而页表地址保存在 pde 中。一个 pte 代表一个物理页，物理页是 4KB 大小，一个页表中可支持 1024 个 pte，故一个页表最大支持 4MB 内存。由于我们目前已经有了一个页表啦，故在 4MB（0x0～0x3ff000）的范围内新增 pte 时，只要申请个物理页并将此物理页的物理地址写入新的 pte 即可，无需再做额外操作。可是，当我们访问的虚拟地址超过了此范围时，比如 0x400000，这不仅是添加 pte 的问题，同时还要申请个物理页来新建页表，同时将用作页表的物理页地址写入页目录表中的第 1 个页目录项 pde 中。也就是说，只要新增的虚拟地址是 4MB 的整数倍时，就一定要申请两个物理页，一个物理页作为新的页表，同时在页目录表中创建个新的 pde，并把此物理页的物理地址写入此 pde。另一个物理页作为普通的物理页，同时在新建的页表中添加个新的 pte，并把此物理页的物理地址写入此 pte。

因此，在添加一个页表项 pte 时，我们务必要判断该 pte 所在的页表是否存在。在第 96 行就是在判断页表项中的 P 位，如果此位为 1，则表示页目录项已存在，不需要再建立。

如果页表已存在，接着在第 99 行判断 pte 是否存在，按理说申请新的地址时其所对应的 pte 不会存在，但为了排除异常的情况，在此还是多判断一下比较放心。如果 pte 不存在，就将物理页的物理地址及相关属性写到此 pte 中，即代码“*pte = (page_phyaddr | PG_US_U | PG_RW_W | PG_P_1)”，这样 vaddr 对应的 pte 就被映射到物理地址 page_phyaddr 上，并添加了属性 US=1，RW=1，P=1，各属性意义在前面介绍头文件时已述。

如果在上面的判断中发现 pde 不存在时，需要申请个新的物理页来创建新的页表，因此第 107 行通过调用“palloc(&kernel_pool)”申请新的物理页并将地址保存在变量 pde_phyaddr 中。随后将新物理页的物理地址 pde_phyaddr 和相关属性写入此 pde 中，即代码“*pde = (pde_phyaddr | PG_US_U | PG_RW_W | PG_P_1)”，属性同样是 US=1，RW=1，P=1。

第 116 行是调用 memset 函数对刚刚申请的物理页初始化为 0，这里面还是有些陈旧数据的。

在上面有几行注释，大概意思是：由于此物理页要被用作页表，所以必须要保证此物理页中的内容是“干净”的，否则一些陈旧数据会让此页表混乱。举个例子，比如此时的页表是新创建的，其中应该没有任何 pte，但此物理页在之前可能已被分配并使用过，用户或内核程序在里面已经写了脏数据，内存管理系统在回收该物理页的时候并没有对其清 0，因为回收时清 0 是低效的，物理页在回收后未必会再分配出去，所以在分配的时候对其清 0 较高效。这个脏数据可能“恰好”形成 pte，其实我用这个“恰好”只是想表达一种“无心插柳柳成荫”的情形，对于这种“形成 pte”的情况并不是偶然的，而是“经常”。因为处理器就是把页表中每 4 字节的数据视为 pte，而各属性位无论取值如何都有相应的意义，所以不管 pte 是否合理，一律全盘接受。这就像遇到了女神，怎么看都顺眼，“缺点、任性”都成了“个性”。

这样形成的 pte 风格迥异，也许 P 位为 1，也许 pte 的高 20 位有数值，这就会导致新的页表中平白无故地多了好多页表项，页表就会“臃肿且破烂不堪”。您可以试一下，如果此处不对用作新页表的物理页清 0 的话，在 bochs 中用 info tab 来查看页表中的映射关系时，能输出好几屏的内容，而且全不是自己添加的，想想就醉了，您懂的，小弟我就深受其苦。

为了初始化物理页，光知道物理页的物理地址是不行的，毕竟在分页机制下一切都是通过虚拟地址来访问的，因此必须得知道该物理页对应的虚拟地址。

既然此物理页用作页表，也就是它会被写入某个 pde 中，咱们先看看现有的变量能否解决。

在本函数开头的指针变量 pde，其值是虚拟地址 vaddr 所在 pde 的虚拟地址，如果对其用*取值，如*pde，其结果是该 pde 中的值，即 4 字节的页目录项的内容：第 0 位为 P 位……高 20 位为页表的物理地址等。得到了物理地址也白搭，我们知道，必须要通过虚拟地址访问，因此，指针 pde 似乎帮助不大。

咱们再看看指针变量 pte，pte 是 vaddr 所在页表项 pte 的虚拟地址，*pte 的值是 vaddr 所在的页表项 pte 的内容，其中高 20 位是普通物理页的物理地址……也依然是物理地址，似乎指针 pte 也并没有什么用，甚至感觉天空也跟着黯淡了许多。

但是……

pte 是 vaddr 所在的页表项 pte 的虚拟地址，它就是 pte_ptr 返回的结果 new_vaddr，new_vaddr 的低 12 位是用来在其高 20 位所定位到的页表中做 pte 的偏移量，也就是说，如果 new_vaddr 的低 12 位为 0，访问到的则是页表的起始虚拟地址，而我们要初始化的物理页就是用作此页表。（不知怎么地，天空又出现了一缕阳光。）言外之意是将 new_vaddr 的低 12 位清 0，就是说 pte 的偏移量不要了，而高 20 位保留，也就是如代码“(int)pte & 0xfffff000”所示，这样就得到了这个 vaddr 所在页表的虚拟地址，也就是刚刚申请的新物理页的虚拟地址。这样就意味着可以对其用 memset 清 0 了，于是用此行代码对该物理页做清 0 操作，即“memset((void*)((int)pte & 0xfffff000)，0，PG_SIZE);”。

接着在本函数的最后，为 vaddr 对应的 pte 赋值，也就是把物理页地址和属性写进去，即“*pte = (page_phyaddr | PG_US_U | PG_RW_W | PG_P_1)”。

有关 page_table_add 函数的介绍至此结束，咱们继续看。

第 124 行的 malloc_page 函数接受 2 个参数，一个是 pf，用来指明内存池，另外一个是 pg_cnt，用来指明页数，此函数的功能是在 pf 所指向的内存池中分配 pg_cnt 个页，成功则返回起始虚拟地址，失败时返回 NULL。

在函数的开头，有一句“ASSERT(pg_cnt > 0 && pg_cnt < 3840)”，它用来监督申请的内存页数 pg_cnt

是否超过了物理内存池的容量。内核和用户空间各约 16MB 空间，保守起见用 15MB 来限制，申请的内存页数要小于内存池大小，即

pg_cnt<15*1024*1024/4096 = 3840 页。

其实此函数是申请虚拟地址，然后为此虚拟地址分配物理地址，并在页表中建立好虚拟地址到物理地址的映射，相当于干了三件事，步骤如下。

（1）通过 vaddr_get 在虚拟内存池中申请虚拟地址。

（2）通过 palloc 在物理内存池中申请物理页。

（3）通过 page_table_add 将以上两步得到的虚拟地址和物理地址在页表中完成映射。

您看，malloc_page 就是以上三个函数的封装。

按照上面的步骤，第 131～134 行先申请虚拟地址，如果失败就返回 NULL。

第 137 行判断要用的内存池属于内核，还是用户，下面要在相应的内存池中分配物理页。

第 140～147 行是循环为虚拟页分配物理页并在页表中建立映射关系。

第 141 行调用 palloc 在相应内存池中申请物理页，物理页地址保存在指针变量 page_phyaddr 中，如果失败（返回值为 NULL）则通过“return NULL”返回。

这里还有些工作没完成，当申请物理页失败时，应该将曾经已申请成功的虚拟地址和物理地址全部回滚，虽然地址还未使用，但虚拟内存池的位图已经被修改了，如果物理内存池的位图也被修改过，还要再把物理地址回滚。由于这部分属于地址回收的功能，不属于本节的内容，待将来实现内存回收时再补充吧。若物理页申请成功，再调用“page_table_add((void*)vaddr, page_phyaddr)”将虚拟地址 vaddr 映射为物理地址 page_phyaddr。随后通过“vaddr += PG_SIZE”将 vaddr 更新为下一个虚拟页，继续下一个循环的申请物理页和页表映射。

当处理完 pg_cnt 个页后，通过“return vaddr_start”将分配的起始虚拟地址返回。

malloc_page 函数就介绍完了，不知有没有同学感到疑惑，为什么要将第（1）步和第（2）、（3）两步拆开，而未放在同一个大的循环中？

原因是这样的，虚拟地址是连续的，但物理地址可能连续，也可能不连续，因此第 1 步中可以一次性申请 pg_cnt 个虚拟页。成功申请之后，根据申请的页数，通过循环依次为每一个虚拟页申请物理页，再将它们在页表中依次映射关联。

最后一个函数是 get_kernel_pages，它只接受一个参数申请的页数 pg_cnt，函数功能是从内核物理内存池中申请 pg_cnt 页内存，成功则返回其虚拟地址，失败则返回 NULL。

在其内部是调用 malloc_page 来实现的，返回的虚拟地址保存在变量 vaddr 中，之后再通过 memset 将此页清 0。由于 malloc_page 返回的是虚拟地址，因此可以直接将 vaddr 作为 memset 的参数，由于虚拟地址是连续的，所以置 0 的字节数直接用 pg_cnt 乘以 PG_SIZE。

申请成功通过 return vaddr 返回分配的虚拟地址。

有关本节内存管理的内容就介绍完了，现在需要测试一下，我们在 main.c 中添加申请内存的代码试试看，见代码 8-13。

代码 8-13　（project/c8/e/kernel/main.c）

```
 1 #include "print.h"
 2 #include "init.h"
 3 #include "memory.h"
 4 int main(void) {
 5     put_str("I am kernel\n");
 6     init_all();
 7
 8     void* addr = get_kernel_pages(3);
 9     put_str("\n get_kernel_page start vaddr is ");
10     put_int((uint32_t)addr);
11     put_str("\n");
12
13     while(1);
14     return 0;
15 }
```

在第 8～11 行通过 get_kernel_pages(3)申请了 3 个物理页。代码没什么可说的，编译链接后写入虚拟硬盘。咱们在 bochs 中运行试试看效果，如图 8-21 所示。

```
I am kernel
init_all
idt_init start
   idt_desc_init done
   pic_init done
idt_init done
timer_init start
timer_init done
mem_init start
   mem_pool_init start
      kernel_pool_bitmap_start:C009A000 kernel_pool_phy_addr_start:200000
      user_pool_bitmap_start:C009A1E0 user_pool_phy_addr_start:1100000
   mem_pool_init done
mem_init done

 get_kernel_page start vaddr is C0100000
CTRL + 3rd button enables mouse  NUM  CAPS  SCRL  HD:0-M HD:0-S
```

▲图 8-21　bochs 运行结果

如图 8-21 所示，屏幕上如预期打印了我们想要的信息，其中用两个方框标出了两个重点信息：上面的框是内核物理内存池的起始物理地址，为 0x200000，下面的框是在 main.c 中申请的虚拟地址。

咱们继续在 bochs 控制台中验证一下，看看以上所有代码的效果，如图 8-22 所示。

图 8-22 中，画下画线的部分是调试命令，方框中的是输出信息。

```
work@localhost:~/my_workspace/bochs
File Edit View Search Terminal Help
<bochs:2> info tab
cr3: 0x000000100000
0x00000000-0x000fffff -> 0x000000000000-0x0000000fffff
0x00100000-0x00102fff -> 0x000000200000-0x000000202fff
0xc0000000-0xc00fffff -> 0x000000000000-0x0000000fffff
0xc0100000-0xc0102fff -> 0x000000200000-0x000000202fff
0xffc00000-0xffc00fff -> 0x000000101000-0x000000101fff
0xfff00000-0xffffefff -> 0x000000101000-0x0000001fffff
0xfffff000-0xffffffff -> 0x000000100000-0x000000100fff
<bochs:3> page 0xc0100000
 PDE: 0x0000000000101067    ps         A pcd pwt U W P
 PTE: 0x0000000000200067       g pat D A pcd pwt U W P
linear page 0xc0100000 maps to physical page 0x000000200000
<bochs:4> page 0xc0101000
 PDE: 0x0000000000101067    ps         A pcd pwt U W P
 PTE: 0x0000000000201067       g pat D A pcd pwt U W P
linear page 0xc0101000 maps to physical page 0x000000201000
<bochs:5> page 0xc0102000
 PDE: 0x0000000000101067    ps         A pcd pwt U W P
 PTE: 0x0000000000202067       g pat D A pcd pwt U W P
linear page 0xc0102000 maps to physical page 0x000000202000
<bochs:6> x 0xc009a000
[bochs]:
0xc009a000 <bogus+       0>:    0x00000007
<bochs:7>
```

▲图 8-22　bochs 控制台

最上面我们先用 info tab 查看了页表中虚拟地址与物理地址的映射关系，在其后面的最大的长方框中是映射关系，左边是虚拟地址的范围，右边是所映射的物理地址。在 bochs 的输出中，无论地址是虚拟地址，还是物理地址，只要其是连续的，bochs 就会合并到一起输出。因此，虚拟地址范围 0xc0100000～0xc0102fff 所映射的物理地址范围是 0x200000～0x202fff。

为验证下各个虚拟地址的映射，用 page 指令分别对各个虚拟页查看其映射到的物理页，大伙儿自己对照着三个命令的输出验证吧。左边的方框是虚拟地址，右边的方框是物理地址。

最后通过 x/10 0xc009a000 查看了内核内存池的位图所在地址，图中最下面的三行是此命令的输出，我们看最后一个框，其内容是 0x00000007，这说明低 3 位都是 1，这是因为咱们申请了 3 个页，位图的变化与预期符合。

本节真是有点长，不过总算结束了，有关内存管理的部分咱们先告一段落，待今后把其他相关的内容都介绍完后咱们再回来继续丰富它的功能。

兄弟们辛苦了，本章任务完成，下一章再见。

第 9 章　线程

9.1 实现内核线程

在我们学习操作系统课程之前就知道，为了充分利用计算机硬件资源，要让计算机尽可能“同时”多做一些工作。“工作”是由处理器执行某段程序代码来完成的，这段程序代码就称为进程，一个进程可以完成一项工作。有的时候，工作往往并不简单，它由多个“子工作”组成，因此，我们需要让进程尽可能“同时”多做一些子工作，这些子工作也得由程序代码完成，完成这些子工作的程序就是线程。

既然是要求同时去做计算机中的“工作”，因此工作无论大小，都是独立的控制执行流，需要处理器单独去执行，所以进程和线程是最基本的执行单元。

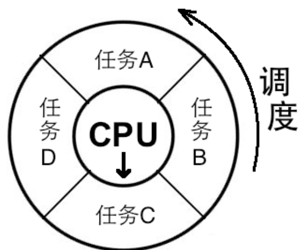
线程和进程咱们要分两部分实现，本节咱们先开始线程之旅。

9.1.1　执行流

过去，计算机只有 1 个处理器（当然，即使是现在，计算机中的处理器数量也不是无限的，一般是 8 核左右），在其上运行的系统也是单任务操作系统，即不管有多少个任务，任务的执行都是串行的，一个任务彻底执行完成后才能开始下一个任务。

假如有任务 A，它的执行要耗时一天（一点都不夸张，在流量较大的业务中有这样的脚本任务，经常跑个一天半天的），任务 B 的执行仅需要 2 分钟，如果任务 A 先上处理器上运行，似乎会严重滞后其他的任务执行计划，用户往往是急躁的，为了执行一个 2 分钟的任务要等上一天，这怎么得了。

于是，在处理器数量不变的情况下，多任务操作系统出现了，它采用了一种称为多道程序设计的方式，使处理器在所有任务之间来回切换，这样就给用户一种所有任务并行运行的错觉，这称为“伪并行”，毕竟在任意时刻，处理器只会执行某一个任务，处理器会落到哪些任务上执行，这是由操作系统中的任务调度器决定的，如图 9-1 所示，处理器固定在圆心，任务就像轮盘一样，由任务调度器把任务转动到处理器的箭头处，这就表示上 CPU 运行。



▲图 9-1　伪并行调度

按理说，一个处理器任意时刻只能执行一个任务，真正的并行是指多个处理器同时工作，一台计算机的并行能力取决于其物理处理器的数量。也就是说，目前本来只有 1 个处理器，但现在非要让其兼顾所有的任务，唯一的做法是只能让每个任务各在处理器上执行一小会儿，然后再换下一个任务上处理器，直到所有任务都执行完毕。

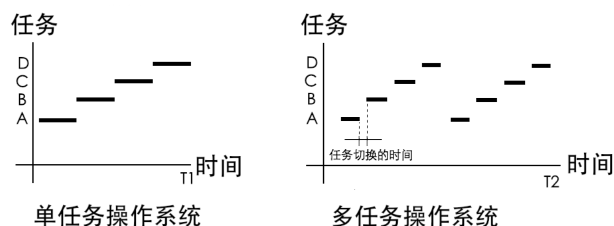
以上的任务轮转工作就是由任务调度器来完成的，什么是任务调度器呢？

简单来说，任务调度器就是操作系统中用于把任务轮流调度上处理器运行的一个软件模块，它是操作系统的一部分。调度器在内核中维护一个任务表（也称进程表、线程表或调度表），然后按照一定的算法，从任务表中选择一个任务，然后把该任务放到处理器上运行，当任务运行的时间片到期后，再从任务表中找另外一个任务放到处理器上运行，周而复始，让任务表中的所有任务都有机会运行。正是因为有了调度器，多任务操作系统才能得以实现，它是多任务系统的核心，它的好坏直接影响了系统的效率。

这种伪并行的好处是降低了任务的平均响应时间，通俗点说，就是让那些执行时间短的任务不会因为“后到”而不得不等前面“先来”的且执行时间很长的程序执行完后才能获得执行的机会，整体上“显得”快了很多。当然这和调度算法有关，这里所说的调度算法是较为“公正”的时间片轮转算法，也称为轮询。

那多任务就没有什么弊端吗？

对于所有任务来说，在不考虑阻塞的情况下，无论是在哪种系统上，它们“自身指令”总共的执行时间之和应该是一致的。但是，在多任务系统中，任务切换是软件完成的，切换工作本身必然要消耗处理器周期，因此所有任务的总共执行时间反而更长了，如图 9-2 所示，四个任务：A、B、C、D 执行的总时间，在多任务操作系统上的时间更长，其中右图中虚线的部分是任务切换的时间成本。



▲图 9-2 多任务系统任务执行时间较长

虽然所有任务总的执行时间变长了一些，但能够让后面紧急的任务及时完成，这点代价算起来也算是微不足道了。

任务并行这是用软件来切换任务模拟出来的假象，处理器根本就不在乎切换任务这回事，不过话不能说得这么绝对，其实处理器原生支持“任务”并提供了相关的结构、方法，比如 TSS 结构和任务门，因此它知道何时发生了任务切换，只是咱们没用它提供的方法，当然这是后话，介绍到后面时您就清楚了，反正它不关心咱们眼中的任务切换是在干吗，甚至还以为这是任务的一部分呢。因为处理器只知道加电后按照程序计数器中的地址不断地执行下去，在不断执行的过程中，我们把程序计数器中的下一条指令地址所组成的执行轨迹称为程序的控制执行流，让我们再深入描述一下。执行流就是一段逻辑上独立的指令区域，是人为给处理器安排的处理单元。指令是具备“能动性”的数据，因此只有指令才有“执行”的能力，它相当于是动作的发出者，由它指导处理器产生相应的行为。指令是由处理器来执行的，它引领处理器“前进”的方向，用“流”来表示处理器中程序计数器的航向，借此比喻处理器依次把此区域中的指令执行完后，所形成的像河流一样曲直不一的执行轨迹、执行路径（由顺序执行指令及跳转指令导致）。

执行流对应于代码，大到可以是整个程序文件，即进程，小到可以是一个功能独立的代码块，即函数，而线程本质上就是函数。

执行流是独立的，它的独立性体现在每个执行流都有自己的栈、一套自己的寄存器映像和内存资源，这是 Intel 处理器在硬件上规定的，其实这正是执行流的上下文环境。因此，我们要想构造一个执行流，就要为其提供这一整套的资源。不知道我说清楚了没有，我的意思是其实任何代码块，无论大小都可以独立成为执行流，只要在它运行的时候，我们提前准备好它所依赖的上下文环境就行，这个上下文环境就是它所使用的寄存器映像、栈、内存等资源。

成为独立的执行流有什么用呢？这可有大用了，在任务调度器的眼里，只有执行流才是调度单元，即处理器上运行的每个任务都是调度器给分配的执行流，只要成为执行流就能够独立上处理器运行了，也就是说处理器会专门运行执行流中的指令。

也许您要问了，程序中哪些才是独立的执行流呢？其实我们早已经耳熟能详了，执行流就是我们要介绍的线程和进程。

说了这么多，我们软件中所做的任务切换，本质上就是改变了处理器中程序计数器的指向，即改变了处理器的“执行流”。

任务只是人为划分的、逻辑上的概念，人们把一个个的执行单元称为任务，我们所说的执行单元就是这些彼此独立的执行流，因此，独立的执行流成了调度器的调度单元，并使之成为了处理器的基本执行单位。

好啦，本节先到这，兄弟们下节再见。

9.1.2 线程到底是什么

说实话，本节我考虑了几次要不要写，毕竟大家工作中线程用得很频繁，线程到底是什么，我想大伙儿比我还清楚。但今天有朋友和我聊起他用多线程解决了工作中的问题，大概内容是他用线程实现了并发同步代码的功能。其实这根本用不着使用线程来做这事，完全可以通过把 N 个同步脚本放在后台“并行”

来解决。造成这种认识的本质原因就是不明白线程的原理，不会灵活运用其他更简单的方法来做线程可以完成的工作。因此，下面和大伙说下我个人对线程的理解。

初次接触线程是在高级语言中，解铃还须系铃人，咱们还是用高级语言来解释它。要不咱们先看看实际代码吧，回忆下线程是怎么用的。

代码 thread_test.c

```
 1 #include<stdio.h>
 2 #include<pthread.h>
 3
 4 void* thread_func(void* _arg) {
 5    unsigned int * arg = _arg;
 6    printf(" new thread: my tid is %u\n", *arg);
 7 }
 8
 9 void main() {
10    pthread_t new_thread_id;
11    pthread_create(&new_thread_id, NULL, thread_func, &new_thread_id);
12    printf("main thread: my tid is %u\n", pthread_self());
13    usleep(100);
14 }
```

这个 thread_test.c 还是蛮简单的，我们在第 2 行包含了 pthread.h，这是 POSIX 版本线程库。第 11 行利用了 pthread_create 函数创建线程，此函数的原型是：

```
int pthread_create (pthread_t *__restrict __newthread,
__const pthread_attr_t *__restrict __attr,
void *(*__start_routine) (void *),
void *__restrict __arg) __THROW __nonnull ((1, 3));
```

此函数接受 4 个参数。

第 1 个参数__newthread 用于存储新创建线程的 id，也就是 tid，这里保存在 pthread_t 类型的变量 new_thread_id 中。

第 2 个参数__attr 用于指定线程的类型，我们这里就用默认类型就好，因此实参是 NULL。

第 3 个参数__start_routine 是个函数指针，确切地说是个返回值为 void*、参数为 void*的函数指针，用来指定线程中所调用的函数的地址，或者说是在线程中运行的函数的地址。这里的实参就是在上面定义的函数 thread_func，也就是说让新创建的线程去调用执行 thread_func 函数。

第 4 个参数__arg，它是用来配合第 3 个参数的，是给在线程中运行的函数__start_routine 的参数，我们此处把 new_thread_id 传给 thread_func。注意，由于给__start_routine 函数做参数的只有这一个形参，当参数多于一个时，最好把参数封装为一个结构体，把此结构体地址传给__arg，然后在__start_routine 指向的函数体中再去解析参数。

pthread_create 函数的返回值若为 0，则表示创建线程成功，否则就表示出错码。

您看，咱们为了简单，也没去判断 pthread_create 的返回值，权当一定会成功，去除了不相关的东西，只剩下赤裸裸的真相。然后在主线程 main 和新线程 thread_func 中分别打印自己的 tid。由于不清楚新线程是否是在主线程结束之前被调用，因此在主线程 main 的最后调用了 usleep 函数使其阻塞 100 微秒。

把 thread_test.c 编译，运行结果如图 9-3 所示。

```
[work@localhost thread]$ gcc -o thread_test.bin -lpthread thread_test.c
[work@localhost thread]$ ./thread_test.bin
main thread: my tid is 3078870720
 new thread: my tid is 3078867824
[work@localhost thread]$
```

▲图 9-3 线程测试

如果您是第一次接触到线程，通过这个简单的例子，不知道您是否看出点什么端倪没有，线程其实就是运行一段函数的载体。

在高级语言中，线程是运行函数的另一种方式，也就是说，构建一套线程方法，让函数在此线程中被

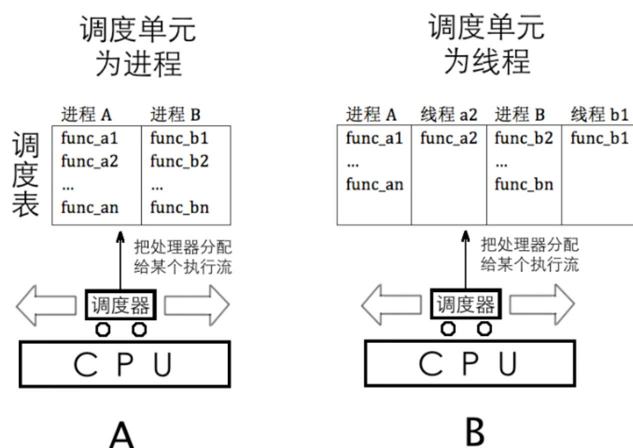
调用，然后处理器去执行这个函数，因此线程实际的功能就是相当于调用了这个函数，从而让函数执行。

那它和普通的函数调用有什么区别呢？

要回答这个问题，这涉及到调度器维护的线程表或进程表了，这属于调度单元的问题。调度器每次安排一个“执行流”上处理器，执行流肯定是独立的，独立的意思就是它有自己的一套寄存器映像，有自己的栈，也就是有独立的上下文件环境。

之前在介绍执行流的时候强调过了，其实任何代码块都可以独立，只要在它运行的时候，我们给它准备好它所依赖的上下文环境就行，这个上下文环境就是它所使用的寄存器映像和栈等资源。只要是独立的执行流就可以被调度器视为一个调度单元，就可以享受处理器的单独服务。我们要做的就是：给任何想单独上处理器的代码块准备好它所依赖的上下文环境，从而使其具备独立性，使之成为执行流，即调度单元。

在一般的函数调用中，它是随着此函数所在的调度单元（执行流）一块上处理器运行的，这个调度单元也许是整个进程，也可能是其他的线程，总之是混在更大的执行流中被“夹杂着、稍带着”执行的，甚至有可能还未执行到此函数它的时间片就到了，从而被换下了处理器，您懂的，处理器毕竟不是专门去执行此函数，说是被顺便执行一点不夸张，因为调度单元是被调度器安排上处理器的，此函数不是单独的调度单元，人家调度器眼里看不到此函数，当然不能专门运行它了。此情况如图 9-4A 所示。



▲图 9-4 线程的作用

图 9-4 中，调度器决定把处理器分配给哪个任务，因此它将左右滑动，选择它认为合适的调度单元，图 9-4A 所示的调度单元是进程级别。

线程是一套机制，此机制可以为一般的代码块创造它所依赖的上下文环境，从而让代码块具有独立性，因此在原理上线程能使一段函数成为调度单元（或称为执行流），使函数能被调度器“认可”，从而能够被专门调度到处理器上执行。这样，函数就可以被加入到线程表中作为调度器的调度单元，从而有机会单独获得处理器资源，也就是说，处理器不是把线程中调用的函数和其他指令混在一块执行的，或者说不是在执行整个进程时顺便执行了该函数，而是单独、专门执行了此函数。

举个例子，就拿咱们在饭店里点菜来说，一般都是咱们点好了菜让厨师给做。什么是菜？饭店里有个规矩，只要是盛在单独的器皿中，如碟子或盘子，就可以成为一道菜来卖，因此就可以把菜名放到饭店的大菜单中。比如咱们拿着菜单，从中点了炒宫保鸡丁这道菜，咱们顾客相当于调度器，菜单相当于进程表，是咱们选择了让厨师烹饪哪道菜。厨师在烹饪过程中，根据菜谱，在菜里加了鸡肉丁、花生米等，此时厨师就扮演处理器的角色，炒菜的过程就相当于进程。但您就是喜欢吃花生，每次上这道菜的时候会先挑花生来吃，这里的花生便是厨师在做宫保鸡丁时“稍带着”加进去的，它相当于进程中的部分代码，虽然它也是宫保鸡丁的配料，但它毕竟只是组成部分而已，此时可以理解为若厨师做了宫保鸡丁这道菜，就要加入少量的花生做配料，而且是在炒菜的某一工序中加进去的，通常就是抓一把花生米放进锅里，此过程很短暂，也就是秒秒钟的事。此时的情景类似于函数在进程中被调用一样，是顺便执行，而不是专门执行，而且执行时间是有限的，一般情况下只占进程时间片的一小部分。既然您喜欢吃花生，为了让您吃个爽快，那咱们专门点一盘花生米就行了，由于花生米是用盘子盛上来的（此处的盘子相当于线程），这时的花生就从配料变成了一道菜（调度单元），厨师用专门的时间烹饪这道菜。因此，同样都是花生，但就是因为有了盘子（线程），使其变成了一道可以被顾客（调度器）选择的菜（调度单元），厨师（处理器）用在烹饪上的时间长了，做的量还多了，此情景就是函数以线程的形式变成了调度单元，被加入到调度器的线程表，使调度器可以看到该函数，从而使该函数单独得到执行。因此，在这个例子中，线程是用于盛菜的盘子，盘子+菜便是调度单元。

总之，在线程中调用函数是让所运行的函数能够以调度单元的身份独立上处理器运行，当函数可以独立运行时，就会有更大的好处，那就是可以让程序中的多个函数（执行流）以并行的方式运行（当然是伪

并行），为程序提速。

9.1.3　进程与线程的关系、区别简述

程序是指静态的、存储在文件系统上、尚未运行的指令代码，它是实际运行时程序的映像。

进程是指正在运行的程序，即进行中的程序，程序必须在获得运行所需要的各类资源后才能成为进程，资源包括进程所使用的栈，使用的寄存器等。

对于处理器来说，进程是一种控制流集合，集合中至少包含一条执行流，执行流之间是相互独立的，但它们共享进程的所有资源，它们是处理器的执行单位，或者称为调度单位，它们就是线程。

可以认为，线程是在进程基础之上的二次并发。

按照进程中线程数量划分，进程分为单线程进程和多线程进程两种。我们平时所写的程序，如果其中未“显式”创建线程，它就属于单线程进程，这就是我们平时所指的“传统型”的进程，否则就属于多线程进程。对于这种传统型进程，不知道大伙儿是否觉得奇怪：居然里面还有 1 个线程？也许您的反应是：“这简直颠覆了我对进程的认知……”是的，很多人都这么想，但是当您意识到线程仅仅是个执行流，而任何程序至少都有一个执行流时，您对此还感到奇怪吗？此处觉得诧异也不要紧，在后面会渐渐水落石出。

举个例子，进程与线程的关系就像公司部门和部门内的人员，部门要完成一个项目，必须要发动部门内所有人员的力量，部门中的每个人都要为此项目承担不同的工作。对于公司来说，公司的战略要落实到各个部门上，公司的业绩要由各部门产出，因此，部门就是一个进程，部门的任务便是要完成的项目，也就是进程运行的结果。部门内人员是具体做事的执行单位，项目被拆分成不同的子任务分派给个人，他们相当于线程，整个项目需要这些人员（线程）共同协作，直到项目完成，也就是进程运行结束。部门（进程）中人员（线程）可多可少，但至少得有一个人（线程）才行，这样部门（进程）才有存在的必要，否则没有人（线程）来干活，部门（进程）就不存在。

我们在操作系统课程中学过，每个进程都运行在自己的地址空间中，话说有内存空间才能存储资源，因此进程拥有此程序运行所需的全部资源。默认情况下进程中只有一个执行流，即一个进程只能干一件事。有些情况下，我们需要在一个地址空间中存在多个执行流，即让进程同时“并行”做很多事，这多个执行流指的就是线程。执行流就是调度器的调度单位，是处理器的执行单元，线程在此方面和进程的行为是一致的，只不过线程不包括位于进程中的、自己所需要的资源，言外之意是线程没有自己独享的地址空间，没空间就无法存储自己的资源，所以线程必须“活”在进程的世界里，借助进程空间中的资源运行。因此，线程和进程比，进程拥有整个地址空间，从而拥有全部资源，线程没有自己的地址空间，因此没有任何属于自己的资源，需要借助进程的资源“生存”，所以线程被称为轻量级进程（照理说，进程被称为重量级线程也未尝不可）。进程和线程都是执行流，它们都具备独立寄存器资源和独立的栈空间，因此线程也可以像进程那样调用其他函数。似乎这么说还是显得没什么“干货”，别急，咱们慢慢解释。

线程仅仅是个执行流，并不是什么高深莫测的东西，它只是被一些线程实现机制增加了神秘感。比如像 POSIX 线程库中的 pthread_create 函数，它的功能是用来创建线程，传给此函数的第三个实参必须是一个事先定义好的函数，这个作为参数的函数就是我们所说的代码块，也就是前面所解释的“执行流”。可见，线程创建函数 pthread_create 仅仅是创建执行流的一种方式而已，没什么神奇的，大伙儿要透过现象看本质。

在显式创建了线程之后，任务调度器就可以把它对应的代码块从进程中分离出来单独调度上处理器执行了，否则调度器会把整个进程当成一个大的执行流，也可以说是把整个进程当成一个线程，从头到尾依次执行下去。

线程是在进程之后才提出的概念，在没有线程之前，进程就是理所当然的执行流，或者说进程只是一个大的执行流（也许执行流没有大小之分，但有数量之别）。在有了线程的概念后（仅仅是在名词概念之后，其实线程这玩意一直存在，后面会提到），执行流便专指粒度更细的线程，因此线程是最小的执行单元。处理器执行任何程序，其过程都是一步步跟随程序中下一步要执行的指令，所以说程序都有执行流，若未显式创建线程，则当前进程中的指令自然也是执行流，也就是只存在一个线程喽。因此纯粹的进程实际上就相当于单一线程的进程，也就是前面所说的单线程进程。进程中若显式创建了多个线程时，就会有多个执行流，也就是多线程进程。

以上所述也许和您之前对进程和线程的印象有些出入，总之，我要表达的是线程就是执行流，不是说只

有“显式”创建的线程才叫线程，线程是后来提出的概念名词，其实质上就是一段引导处理器执行的、具有能动性的代码，而它早就存在很久了，只不过之前程序（我们现在称之为进程）中只有一段执行流而已，当现在的程序中存在多个执行流时，我们用这个新名词“线程”来称呼它们。

为什么要有线程这一称呼？

我想大家肯定都知道答案，为了给程序提速，确切地说是给进程提速，因为线程必然属于某一进程，线程要运行必须要有相应的资源，而进程就是这个资源的提供者，因此线程存在于进程之中（这个概念我在后面也会反复阐述）。

利用线程提速，原理就是实现多个执行流的伪并行，如图 9-2 右图所示。任务其实就是执行流，要么是大的执行流——单线程的进程，要么是小的执行流——线程。

进程采用多个执行流和其他进程抢处理器资源，这样就节省了单个进程的总执行时间。

提速的原理很简单，就是想办法让处理器多执行自己进程中的代码，这样进程执行完成得就快。就像虽然图 9-2 右图中有 4 个任务，看似其他任务都是在和自己竞争处理器资源，这影响了任何任务的执行速度。但如果其他任务或者大部分任务都是帮任务 A 做事，任务 A 不就很快执行完成了吗？这就是线程提速的原理之一。假如线程是在内核中实现，此时系统中一共有 2 个任务，进程 A 和进程 B，进程 A 为了提速，创建了 3 个线程，任务调度器中便有了 4 个执行流（不包括主线程），其中有 3 个都属于进程 A，也就是说这调度器把所有任务调度一圈后，进程 A 相当于被处理器执行了三次，而进程 B 只在处理器上运行了一次，进程 A 当然执行得快了。

线程另一个提速的原理是避免了阻塞整个进程，当然这指的是内核级线程的实现，一会儿咱们再细说。

比如当进程因等待用户输入而暂时无法继续运行时，此时操作系统会把整个进程挂起，也就是将其从就绪队列中去除，这样便无法获得执行的机会，等用户输入完成，可以继续执行后，操作系统再将其加入到就绪队列，这样调度器才会重新调度它上处理器运行。然而，并不是进程中所有的部分都依赖于用户的输入，对于那些不依赖于用户输入的代码块，可以为其单独创建一线程来“并行”执行，这样进程的某个执行流阻塞于用户输入时，此进程的另一线程还能运行，还能继续做其他事，相当于给进程提速了，岂不美哉。因此，通常程序员写程序时会把整个任务划分成几个独立的部分，每一部分就用线程来完成，各部分是独立无依赖的，因此这几个线程就可以“并行”运行。

进程和线程同样都是执行流，那它们为什么叫不同的名字？之间有什么区别吗？

其实前面或多或少已经提到过一些了，最初进程中只有一条执行流，大家的想法是程序就应该沿着这条路执行下去，谁也不会给“理所当然”的事情起个名字，只是后来为了让程序提速，进程中的执行流变成两条以上了，为了强调进程中包含不同的程序流（执行流），这才出现了线程的概念。其实在处理器上运行的执行流都是“人为划分的”“逻辑上独立的”程序段，本质上都是一段代码区域，只不过线程是纯粹的执行部分，它运行所需要的资源存储在进程这个大房子中，进程中包含此进程中所有线程使用的资源，因此线程依赖于进程，存在于进程之中，用表达式来表示：进程=线程+资源。

举个例子，比如在饭店里，只要有人点菜，厨房就要开始忙活。厨房就相当于进程，里面有食材和烹饪的锅具等，这些都是资源，在厨房中工作的人有厨师、配菜员、餐具清洁员等，他们都是进程中的线程。比如客人点了一盘鱼香肉丝，厨房中各类角色就要开始并行工作，配菜员开始准备食材，厨师负责烹饪，配菜员和厨师这两个线程是各干各的，但他们只能在厨房里工作，他们出了厨房后，什么都干不了，毕竟他们工作时所用的资源，即食材、锅具等都在厨房里，但他们每个人确实都可以分开工作，都是单独的执行流，最终做出鱼香肉丝的是这些具有能动性的人，而不是锅具食材等静态资源。不知道这样解释，您是否理解了进程和线程的关系。

通过上面这个例子，进程和线程的关系我想您也看出来了：进程拥有整个地址空间，其中包括各种资源，而进程中的所有线程共享同一个地址空间，原因很简单，因为这个地址空间中有线程运行所需要的资源。

由于各个进程都拥有自己的虚拟地址空间，正常情况下它们彼此无法访问到对方的内部，因为进程之间的安全性是由操作系统的分页机制来保证的，只要操作系统不要把相同的物理页分配给多个进程就行了。

但进程内的线程可都是共享这同一地址空间的，它们彼此能“见面”，这就暴露出一个问题，既然进程内的所有线程共享同一个地址空间，也就意味着任意一个线程都可以去访问同一进程内其他线程的数

据，甚至可以改变它们的数据，这样岂不是很危险？同意，是很危险，进程内的所有线程确实有对彼此威胁的可能，这肯定避免不了。但是按理说，进程是由一个人或一个开发团队编写的，团队中全是“自己人”，如果自己都对自己“下手”，那即使是单线程进程也照样不安全，毕竟单线程也能做出类似“自残”的事情，这种安全隐患属于人为意识的问题。

强调下，只有线程才具备能动性，它才是处理器的执行单元，因此它是调度器眼中的调度单位。进程只是个资源整合体，它将进程中所有线程运行时用到资源收集在一起，供进程中的所有线程使用，真正上处理器上运行的其实都叫线程，进程中的线程才是一个个的执行实体、执行流，因此，经调度器送上处理器执行的程序都是线程。

这么说的原因是即使进程中未显式创建线程的话，进程中总会有一个向下执行的方向，即执行流，就是前面所说的单线程进程，如果进程中显式创建了多个线程的话，此进程称为多线程进程。总之线程属于进程之内，进程内必有线程。也就是说，任何进程都有自己的执行流，如果只有一个执行流，该执行流可以称为主线程，因为其他新的线程也要通过此主线程创建。

再看看实际的例子，比如我们所使用的浏览器，它访问网页的时候，必然是同时加载网页中的诸多资源，比如多个图片和多个 CSS 样式并发请求，这样浏览器才不会显得“很卡很慢”。对于处理器来说，浏览器仅仅是个进程而已，它能同时做这么多事，必然是内部多个线程同时“发力”的结果。

说了这么多，举了这么多例子，怎样理解进程与线程的区别与关系呢？

其实它们的关系就是前面所说过的表达式：进程 = 资源 + 线程。

为了表达清楚，我还是拿厨房举例子。比如某饭店能做多种菜肴。厨房 A 专门做中餐，厨房 B 专门做西餐，厨房 C 专门做泰国菜，这三个厨房相当于饭店中三个不同的进程，各自厨房中的工作人员便是具有执行力的线程，工作人员才是提供烹饪“思路”的执行流。

而思路是哪来的呢？肯定是根据现有的资源环境想出的，即环境提供解决办法，也就是线程要想运行，只能依赖于进程提供的资源。就像大伙儿都知道软件 Word 提供的功能是文字处理等，因此您在实际的文字编辑工作中，只能根据 Word 提供的功能来完成自己的工作，而具体用什么方法，取决于您得提前知道 Word 能干什么，支持哪些功能才行。说白了，就是大伙儿想看电影的时候，根本不会想到用 Word 去播放视频文件，因为咱们已经提前知道它不是干这个的。咱们继续回来说厨房的事。厨房工作人员的资源环境是什么？是他们所在的各个厨房，厨房便是厨房内工作人员的资源池，里面准备好了做本菜系中各种菜的食材和工具，因此厨房中的工作人员才能烹饪出美味佳肴。各厨房就是不同的进程，因此每个厨房提供的资源也是不同的，各厨房自己的工作人员就是各厨房内不同的线程，他们提供了烹饪本菜系内某道菜的方法和步骤，因此，各厨房的工作人员（线程）只能在自己所在的厨房（进程）中工作，他们（线程）的烹饪思路仅限于本厨房（进程），即使换了个厨房（进程），也无用武之地（假设他们不会做其他菜系），而且，厨房（进程）里多了这样一个外行（线程），反而会影响整个厨房（进程）的工作。

总结一下：

线程是什么？具有能动性、执行力、独立的代码块。

进程是什么？进程=线程+资源。根据进程内线程的数量，进程可分为。

（1）单线程进程：如果厨房中只有一个工作人员，即配菜、炒菜、洗涮厨具等这几样工作都是一个人做，那么厨房的工作效率必然会很低，因为此时厨房中就一个线程。

（2）多线程进程：厨房人手多了，工作才更高效，因此为配菜、炒菜、洗涮厨具专门配备了 3 个工作人员，也就是说进程内的线程多了。

执行流、调度单位、运行实体等概念都是针对线程而言的，线程才是解决问题的思路、步骤，它是具有能动性的指令，因此只有它才能上处理器运行，即一切执行流其实都是线程，因为任何时候进程中都至少存在一个线程。

进程独自拥有整个地址空间，在这个空间中装有线程运行所需的资源，所以地址空间相当于资源容器，就像鱼缸为鱼提供了水。因此，进程与线程的关系是进程是资源容器，线程是资源使用者。进程与线程的区别是线程没有自己独享的资源，因此没有自己的地址空间，它要依附在进程的地址空间中从而借助进程

的资源运行。说白了就是线程没有自己的页表，而进程有。

另外，由于我们对进程这个词太熟悉了，为方便陈述，以后再提到进程的时候，大家知道我是指单线程进程就好，所以今后的内容介绍咱们还是以名词“进程”为主，本节到此结束。

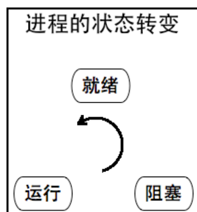
9.1.4　进程、线程的状态

程序在运行时，有可能因为某种情况而无法继续运行，比如某进程的功能是分析日志，它先要读取磁盘，把日志从文件系统中读入内存，之后再从内存中分析日志。访问文件系统需要经过外部设备的操作，比如硬盘，这通常比较耗时，因此在等待 IO 操作的时间内，分析日志的工作是无法进行的，该进程无法做任何事，只能等待。当日志文件从磁盘调入到内存后，进程便准备做日志分析的工作了，之后开始运行日志分析的代码，处理日志，直到完成。

您看，程序从执行到结束的整个过程中，并不是所有阶段都一直开足马力在处理器上运行，有的时候也会由于依赖第三方等“种种无奈”的外在条件而不得不停下来，当这种情况出现时，操作系统就可以把处理器分配给其他线程使用，这样就可以充分利用处理器的宝贵资源了。

为此，操作系统把进程“执行过程”中所经历的不同阶段按状态归为几类，注意，强调的是“执行过程”，意为进程的状态描述的是进程中有关“动作”的执行流部分，即线程，而不包括静止的资源部分。把上述需要等待外界条件的状态称为“阻塞态”，把外界条件成立时，进程可以随时准备运行的状态称为“就绪态”，把正在处理器上运行的进程的状态称为“运行态”。

只要“条件”成立，进程的状态就可以改变，通常这种状态的转变是由操作系统的调度器及相关代码负责的，因为只有它们才知道“条件”是否满足了，如图 9-5 所示。



▲图 9-5　进程的状态变化

进程的状态表示进程从出生到死亡的一系列所处的阶段，操作系统的调度器可以利用它更加高效地管理进程调度。

以上虽然是以进程举例，但实际上已经和大伙儿强调过多次了（希望大家不要嫌我烦，有些陌生的内容需要量变才能到质变），调度器的调度单位是执行流，“状态”描述的也是执行流，而“状态”又主要是给调度器用的，因此“状态”是对所有执行流而言的概念，这里所说的进程状态其实就是指单线程进程中线程的状态，归根结底，状态是描述线程的。

总之，进程或线程等各种执行流都是人为创造的代码块，因此执行流的各种状态也是人为划分的，这些都是操作系统自我管理、自圆其说的一套体系，进程有哪些状态，取决于操作系统对进程的管理方法，这没有定律，咱们也可以创造一套自己的进程状态。

9.1.5　进程的身份证——PCB

大伙儿已经知道现代操作系统都是多任务操作系统，每个任务要被调度到处理器上分时运行，运行一段时间后再被换下来，由调度系统根据调度算法再选下一个线程上处理器，是这回事吧？于是问题接连不断地来了。

（1）要加载一个任务上处理器运行，任务由哪来？也就是说，调度器从哪里才能找到该任务？

（2）即使找到了任务，任务要在系统中运行，其所需要的资源从哪里获得？

（3）即使任务已经变成进程运行了，此进程应该运行多久呢？总不能让其独占处理器吧。

（4）即使知道何时将其换下处理器，那当前进程所使用的这一套资源（寄存器内容）应该存在哪里？

（5）进程被换下的原因是什么？下次调度器还能把它换上处理器运行吗？

（6）前面都说过了，进程独享地址空间，它的地址空间在哪里？

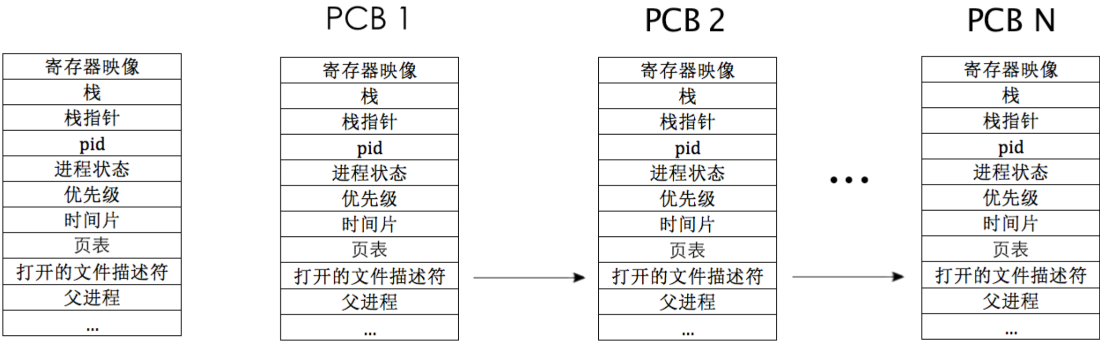
……

以上只是一些调度相关的问题，其实还有其他问题呢，暂且列举到这。

为解决以上问题，操作系统为每个进程提供了一个 PCB，Process Control Block，即程序控制块，它就是进程的身份证，用它来记录与此进程相关的信息，比如进程状态、PID、优先级等。一般 PCB 的结构如图 9-6 所示。

每个进程都有自己的 PCB，所有 PCB 放到一张表格中维护，这就是进程表，调度器可以根据这张表

选择上处理器运行的进程，如图 9-7 所示。PCB 就成了进程表中的“项”，因此，PCB 又可称为进程表项。



▲图 9-6　PCB 结构　　▲图 9-7　进程表

PCB 没有具体的格式，其实际格式取决于操作系统的功能复杂度，以上只是列出了基本该有的内容。

您看，PCB 中包含“进程状态”，它解决了上面第 5 个问题，比如进程状态为阻塞态，下次就不能把它调度到处理器上了。“时间片”解决上面第 3 个问题，当时间片的值为 0 时，表示该进程此次的运行时间到期了，该下 CPU 啦。“页表”解决了上面第 6 个问题，它代表进程的地址空间。还有一些问题没有解决，不急，咱们慢慢来。

PCB 可大可小，尺寸不定，不知道大伙儿注意到没有，在 PCB 的最上面是寄存器映像，它是啥意思呢？“寄存器映像”是用来解决上面第 4 个问题的，即保存进程的“现场”，进程在处理器上运行时，所有寄存器的值都将保存到此处。一般情况下它位于 PCB 的顶端，不过位置也不固定，具体位置取决于 0 级栈指针的位置，总之它会在 PCB 中上下滑动。其实第 4 个问题解决了，第 2 个问题也就一同搞定了，再从 PCB 中把寄存器映像加载到寄存器中就行了。

目前只剩下第 1 个问题没有解决了，其实，要解决此问题，就是要单独维护一个进程表，将所有的 PCB 结构加载到此表中，由调度器直接在进程表中找相应进程的 PCB，从而获取到对应进程的信息，将其寄存器映像加载到处理器后，新进程就开始运行了。

在 PCB 中，还有个“栈”没有说，不过聪明的您一定想到了：进程使用的栈也属于 PCB 的一部分，不过此栈是进程所使用的 0 特权级下内核栈（并不是 3 特权级下的用户栈）。栈在 PCB 中，这听上去有点不可思议，但细想一下还是觉得合理的。和进程相关的所有资源都应该集中放在一起，这样才方便管理。

您看，既然内核栈都要放在 PCB 中，那么，PCB 一般都很大，通常以页为单位，咱们系统比较小，PCB 只占一页。顺便说一句，上面所说的“寄存器映像”的位置并不固定，原因就是“寄存器映像”存储到内核栈中，通常情况下进程或线程被中断时，处理器自动在 TSS 中获取内核栈指针，这通常是 PCB 的顶端，因此通常情况下“寄存器映像”位于 PCB 的顶端。但有时候进程或线程的上下文，也就是“寄存器映像”，并不是在中断发生时保存到栈中的，而是在内核态下工作时，栈指针已经发生了变化时才向栈中保存“寄存器映像”，比如线程主动让出处理器，这时候就得保存线程的现场，此时“寄存器映像”必然就不在 PCB 顶端了。提醒一下，内核态未必都是关中断的状态，可以在开中断下执行内核代码，否则就不会接收时钟中断，进而就不会调用调度器，也就无法进行任务调度了，本章介绍相关内容时大伙儿就会体会到。鉴于“寄存器映像”的位置并不固定，我们在 PCB 中还要维护一个“栈指针”成员，它记录 0 级栈栈顶的位置，借此找到进程或线程的“寄存器映像”。

好啦，PCB 的引入就是为了解决以上几个核心问题，因此 PCB 乃是进程的核心，更多内容还是留待实践中体会吧。

9.1.6　实现线程的两种方式——内核或用户进程

起初，操作系统中只有进程的概念，人们那时候对并发没有太高的要求。后来有些人想提高程序的并发，这才有了线程这一新生事物。任何新生事物在诞生之初都会被小心谨慎地对待，人们提出线程的需求

时，操作系统也抱着“围观”的心态不敢轻举妄动，只能坐看其发展，真正待需求明朗时才会在操作系统一级来实现。想想也是，如果稍微有个新需求就往内核里面塞，内核开发成本很高不说，至少内核中肯定有很多“不切实际”的功能，所以人们还是能够体谅操作系统研发厂商的。

为此，既然不能说服操作系统支持线程，人们只好在用户进程内想办法。所以，线程的实现就有两种方式，要么由操作系统原生支持，用户进程通过系统调用使用线程，要么操作系统不支持线程，由进程自己想办法解决。因此，线程要么在 0 特权级的内核空间中实现，要么在 3 特权级的用户空间实现。

强调一下，这里所说的“在 0 特权级的内核空间中实现线程”，只是说线程机制由内核来提供，并不是说线程中所运行的代码也必须是 0 特权级的内核级代码，也可以是 3 特权级的用户级代码，内核毕竟是为用户进程提供服务的。而“在 3 特权级的用户空间实现线程”，是指线程机制由用户进程自己提供，相当于用户进程除了负责业务外，还要在进程中实现线程调度器，这样一来程序员负担比较重，所以通常情况下很少有程序员愿意在进程中写线程机制，故标准库便提供了用户级线程库，程序员直接使用标准线程库就行了。

用户特权级是 3，因此线程中只能运行自己进程内的代码，即只能同级调用，不能调用 0 特权的内核代码。

总之，无论线程机制是由内核，还是用户进程提供，都是为用户进程服务的，线程中必须可以运行用户的代码。

下面看看由这两类提供方实现的线程各自的优缺点。

线程仅仅是个执行流，在用户空间，还是在内核空间实现它，最大的区别就是线程表在哪里，由谁来调度它上处理器。如果线程在用户空间中实现，线程表就在用户进程中，用户进程就要专门写个线程用作线程调度器，由它来调度进程内部的其他线程。如果线程在内核空间中实现，线程表就在内核中，该线程就会由操作系统的调度器统一调度，无论该线程属于内核，还是用户进程。

下面分别讨论下这两种情况下的实现方式。

1．在用户空间中实现线程

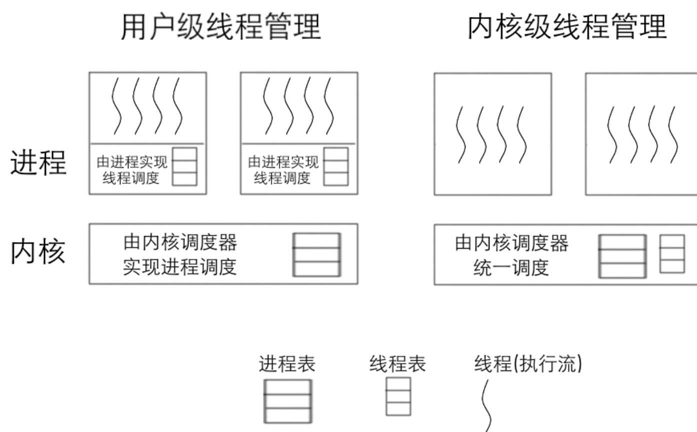
注意，咱们这里讨论的是线程只由用户进程来实现，操作系统中无线程机制。

在用户空间中实现线程的好处是可移植性强，由于是用户级的实现，所以在不支持线程的操作系统上也可以写出完美支持线程的用户程序。

原理很简单，在用户空间中实现线程，操作系统根本就不会意识到线程的存在，因为操作系统调度器只会以整个进程的方式调度，将处理器的使用权交给这个进程，由进程中的调度器自己去协调分配处理器时间。

无论线程在哪里实现，目的都是要到处理器上运行，因此必然要考虑到线程调度的问题，这涉及到调度器及线程表。

如果在用户空间中实现线程，用户线程就要肩负起调度器的责任，因此除了要实现进程内的线程调度器外，还要自己在进程内维护线程表，如图 9-8 中左图所示。



▲图 9-8　线程的两种实现方式

用户进程中，很少有人亲自写线程调度器，因此，一般是某个权威机构发布个用户级线程包，也就是线程

库，开发人员在用户进程中调用此包中的方法去创建线程、结束线程等。线程包中一定存在着线程调度器，而且，线程包中的方法都会与此线程调度器有调用关系，这样当有新线程产生或有线程退出时，线程调度器才会被调用，从而在内部维护的线程表中找出下一个线程上处理器运行。

在用户进程中实现线程有以下优点。

- 线程的调度算法是由用户程序自己实现的，可以根据实现应用情况为某些线程加权调度。
- 将线程的寄存器映像装载到 CPU 时，可以在用户空间完成，即不用陷入到内核态，这样就免去了进入内核时的入栈及出栈操作。

当然，任何事物都有两方面，用户级线程也会有以下缺点。

- 进程中的某个线程若出现了阻塞(通常是由于系统调用造成的)，操作系统不知道进程中存在线程，它以为此进程是传统型进程（单线程进程），因此会将整个进程挂起，即进程中的全部线程都无法运行，得，这下因小失大了。

除非咱们在用户空间中写个包裹函数将系统调用封装起来，在包裹函数里面判断该系统调用是否会造成阻塞（事先知道哪些系统调用会引起阻塞），如果现在不阻塞，则允许马上调用，否则，待该系统调用不阻塞时再调用，即推迟调用。

也许有读者会说：“你之前说过，阻塞是操作系统管理进程的方法，那咱们可不可以改下操作系统，把引起阻塞的系统调用改为非阻塞式的？听上去是可以的，但这有点自相矛盾，在用户空间实现线程不就是为了可移植吗，如果把操作系统改了，移植性的优势从何谈起呢，而且代价有点大不是吗？

- 线程未在内核空间中实现，因此对于操作系统来说，调度器的调度单元是整个进程，并不是进程中的线程，所以时钟中断只能影响进程一级的执行流。当时钟中断发生后，操作系统的调度器只能感知到进程一级的调度实体，它要么把处理器交给进程 A，要么交给进程 B，绝不可能交给进程中的某个线程，也就是说，要想让操作系统调度，操作系统得知道它的存在才行，但由于线程在用户空间中实现，线程属于进程自己的“家务事”，操作系统根本不知道它的存在。这就导致了：如果在用户空间中实现线程，但凡进程中的某个线程开始在处理器上执行后，只要该线程不主动让出处理器，此进程中的其他线程都没机会运行。也就是说，没有保险的机制使线程运行“适时”，即避免单一线程过度使用处理器，而其他线程没有调度的机会。这只能凭借开发人员“人为”地在线程中调用类似 pthread_yield 或 pthread_exit 之类的方法使线程发扬“高风亮节”让出处理器使用权，此类方法通过回调方式触发进程内的线程调度器，让调度器有机会选择进程内的其他线程上处理器运行。重复强调：这里所说的“线程让出处理器使用权”，不是将整个进程的处理器使用权通过操作系统调度器交给其他进程，而是将控制权交给此进程自己的线程调度器，由自己的调度器将处理器使用权交给此进程中的下一个线程，肥水不流外人田嘛。
- 最后，线程在用户空间实现，和在内核空间实现相比，只是在内部调度时少了陷入内核的代价，确实相当于提速，但由于整个进程占据处理器的时间片是有限的，这有限的时间片还要再分给内部的线程，所以每个线程执行的时间片非常非常短暂，再加上进程内线程调度器维护线程表、运行调度算法的时间片消耗，反而抵销了内部调度带来的提速。

2．在内核空间中实现线程

注意，这里所说的“实现线程”是指由内核提供原生线程机制，用户进程中不再单独实现。

个人觉得，线程由内核来实现，进程才真正得到了较大幅度的提速，这是最大的优点，这体现在以下方面。

（1）相比在用户空间中实现线程，内核提供的线程相当于让进程多占了处理器资源，比如系统中运行有进程 A 和一传统型进程 B，此时进程 A 中显式创建了 3 个线程，这样一来，进程 A 加上主线程便有了 4 个线程，加上进程 B，内核调度器眼中便有了 5 个独立的执行流，尽管其中 4 个都属于进程 A，但对调度器来说这 4 个线程和进程一样被调度，因此调度器调度完一圈后，进程 A 使用了 80%的处理器资源，这才是真正的提速。

（2）另一方面的优点是当进程中的某一线程阻塞后，由于线程是由内核空间实现的，操作系统认识线程，所以就只会阻塞这一个线程，此线程所在进程内的其他线程将不受影响，这又相当于提速了。

缺点是用户进程需要通过系统调用陷入内核，这多少增加了一些现场保护的栈操作，这还是会消耗一些处理器时间，但和上面的大幅度提速相比，这不算什么大事。

线程实现的两种方式都讨论完了，咱们选择哪种呢？

梳理下思路，进程中实现线程，咱们要自己写线程调度算法，而且将来用户进程是由用户自己写的，线程机制肯定就不唯一了，或者为了唯一，咱们得专门定个线程库，算啦，还是用内核级线程实现起来较容易，而且还比较快，就定它吧，下一节咱们开始写代码喽。

9.2　在内核空间实现线程

本节咱们要在内核空间实现线程，为了大伙儿易于学习，咱们还是循序渐进，先从最简单的做起，在今后随着新功能的添加，逐步完善。

9.2.1　简单的 PCB 及线程栈的实现

下面咱们先构造 PCB 及其相关的基础部分，我把它定义在 thread.h 中啦，这是咱们新增的两个文件之一，另一个文件当然就是 thread.c，说完了头文件再介绍它。那咱们先看代码 9-1-1。

代码 9-1-1　（project/c9/a/thread/thread.h）

```
 1 #ifndef __THREAD_THREAD_H
 2 #define __THREAD_THREAD_H
 3 #include "stdint.h"
 4
 5 /* 自定义通用函数类型，它将在很多线程函数中作为形参类型 */
 6 typedef void thread_func(void*);
 7
 8 /* 进程或线程的状态 */
 9 enum task_status {
10    TASK_RUNNING,
11    TASK_READY,
12    TASK_BLOCKED,
13    TASK_WAITING,
14    TASK_HANGING,
15    TASK_DIED
16 };
17
18 /***********   中断栈 intr_stack   ***********
19  * 此结构用于中断发生时保护程序（线程或进程）的上下文环境：
20  * 进程或线程被外部中断或软中断打断时，会按照此结构压入上下文
21  * 寄存器，intr_exit 中的出栈操作是此结构的逆操作
22  * 此栈在线程自己的内核栈中位置固定，所在页的最顶端
23 ********************************************/
24 struct intr_stack {
25     uint32_t vec_no;        // kernel.S 宏 VECTOR 中 push %1 压入的中断号
26     uint32_t edi;
27     uint32_t esi;
28     uint32_t ebp;
29     uint32_t esp_dummy;
  // 虽然 pushad 把 esp 也压入，但 esp 是不断变化的，所以会被 popad 忽略
30     uint32_t ebx;
31     uint32_t edx;
32     uint32_t ecx;
33     uint32_t eax;
34     uint32_t gs;
35     uint32_t fs;
36     uint32_t es;
37     uint32_t ds;
38
39 /* 以下由 cpu 从低特权级进入高特权级时压入 */
40     uint32_t err_code;      // err_code 会被压入在 eip 之后
41     void (*eip) (void);
42     uint32_t cs;
43     uint32_t eflags;
44     void* esp;
45     uint32_t ss;
46 };
47
```

在代码 9-1-1 的开头，用 typedef 定义了 thread_func，它用来指定在线程中运行的函数类型。我们在线程中打算运行某段代码（函数）时，需要一个参数来接收该函数的地址，因此这里先定义这个返回值 void 的函数类型，以后在介绍其他函数实现时大家会多次见到它。

接下来用 enum task_status 结构定义了线程的状态，当然这也是进程的状态，进程与线程的区别是它们是否独自拥有地址空间，也就是是否拥有页表，程序的状态都是通用的，因此 enum task_status 结构同样也是进程的状态。

这里先定义了 6 个状态，从 TASK_RUNNING 到 TASK_DIED，但目前只用到小部分，先提前定义好了吧，以后免得老提到它。

接下来用 struct intr_stack 定义了程序的中断栈，无论是进程，还是线程，此结构用于中断发生时保护程序的上下文环境。也就是说，进入中断后，在 kernel.S 中的中断入口程序“intr%1entry”所执行的上下文保护的一系列压栈操作都是压入了此结构中。因此，进程或线程被外部中断或软中断打断时，中断入口程序会按照此结构压入上下文寄存器，所以，kernel.S 中 intr_exit 中的出栈操作便是此结构的逆操作。初始情况下此栈在线程自己的内核栈中位置固定，在 PCB 所在页的最顶端，每次进入中断时就不一定了，如果进入中断时不涉及到特权级变化，它的位置就会在当前的 esp 之下，否则处理器会从 TSS 中获得新的 esp 的值，然后该栈在新的 esp 之下，这是后话，有关 TSS 这方面的内容以后会介绍。

咱们继续看下半部分。

代码 9-1-2　（project/c9/a/thread/thread.h）

```
48 /***********  线程栈 thread_stack  ***********
49   * 线程自己的栈，用于存储线程中待执行的函数
50   * 此结构在线程自己的内核栈中位置不固定，
51   * 仅用在 switch_to 时保存线程环境。
52   * 实际位置取决于实际运行情况。
53   ******************************************/
54  struct thread_stack {
55     uint32_t ebp;
56     uint32_t ebx;
57     uint32_t edi;
58     uint32_t esi;
59
60 /* 线程第一次执行时，eip 指向待调用的函数 kernel_thread
61 其他时候，eip 是指向 switch_to 的返回地址*/
62     void (*eip) (thread_func* func, void* func_arg);
63
64 /*****    以下仅供第一次被调度上 cpu 时使用    ****/
65
66 /* 参数 unused_ret 只为占位置充数为返回地址 */
67     void (*unused_retaddr);
68     thread_func* function;         // 由 kernel_thread 所调用的函数名
69     void* func_arg;                // 由 kernel_thread 所调用的函数所需的参数
70 };
71
72 /* 进程或线程的 pcb，程序控制块 */
73 struct task_struct {
74     uint32_t* self_kstack;         // 各内核线程都用自己的内核栈
75     enum task_status status;
76     uint8_t priority;              // 线程优先级
77     char name[16];
78     uint32_t stack_magic;          //栈的边界标记，用于检测栈的溢出
79 };
80
……以下至结束是函数声明
84 #endif
```

结构体 struct thread_stack 定义了线程栈，此栈有 2 个作用，主要就是体现在第 5 个成员 eip 上。

（1）大家都知道，线程是使函数单独上处理器运行的机制，因此线程肯定得知道要运行哪个函数，首次执行某个函数时，这个栈就用来保存待运行的函数，其中 eip 便是该函数的地址。

（2）将来咱们是用 switch_to 函数实现任务切换，当任务切换时，此 eip 用于保存任务切换后的新任务的返回地址。

虽然这么说有些粗糙，但不看实际代码的话还真说不清楚，还是在后面慢慢体会吧。

那线程栈结构中的前 4 个成员它们是干吗的呢？

```
uint32_t ebp;
uint32_t ebx;
uint32_t edi;
uint32_t esi;
```

这涉及到 ABI 内容了，ABI 是 Application Binary Interface，即应用程序二进制接口，也许部分同学只听说过 API，API 是 Application ProgrammingInterface，即应用程序可编程接口，不过这是库函数和操作系统的系统调用之间的接口。ABI 与此不同，ABI 规定的是更加底层的一套规则，属于编译方面的约定，比如参数如何传递，返回值如何存储，系统调用的实现方式，目标文件格式或数据类型等。只要操作系统和应用程序都遵守同一套 ABI 规则，编译好的应用程序可以无需修改直接在另一套操作系统上运行。

好啦，有关 ABI 的介绍就到这，咱们还是说说这 4 个寄存器的故事。在官方规范 SysV_ABI_386-V4 中有这样一段话，原文如下：

All registers on the Intel386 are global and thus visible to both a calling and a called function. Registers %ebp, %ebx, %edi, %esi, and %esp ‘‘belong’’ to the calling function. In other words, a called function must preserve these registers’ values for its caller. Remaining registers ‘‘belong’’ to the called function. If a calling function wants to preserve such a register value across a function call, it must save the value in its local stack frame.

大概意思是：位于 Intel386 硬件体系上的所有寄存器都具有全局性，因此在函数调用时，这些寄存器对主调函数和被调函数都可见。这 5 个寄存器 ebp、ebx、edi、esi、和 esp 归主调函数所用，其余的寄存器归被调函数所用。换句话说，不管被调函数中是否使用了这 5 个寄存器，在被调函数执行完后，这 5 个寄存器的值不该被改变。因此被调函数必须为主调函数保护好这 5 个寄存器的值，在被调函数运行完之后，这 5 个寄存器的值必须和运行前一样，它必须在自己的栈中存储这些寄存器的值。

以上就是我的大概理解，我怕自己理解有偏差，所以贴出了英文原版，请大家辩证地看。

为什么要强调 ABI 呢？

原因是 C 编译器就是按照这套 ABI 规则来编译 C 程序的，倘若咱们全是用 C 语言来写程序，咱们就不需要考虑 ABI 规则，这些都是编译器考虑的事。C 语言和汇编语言是用不同的编译器来编译的，C 语言代码要先被编译为汇编代码，此汇编代码便是按照 ABI 规则生成的，因此，如果要自己手动写汇编函数，并且此函数要供 C 语言调用的话，咱们也得按照 ABI 的规则去写汇编才行。说到这您肯定明白了，咱们一定是用汇编语言写了个函数，而且是用 C 程序来调用这个汇编函数。

对！确实如此，程序中处处都是伏笔，很多结构都是为了其他函数准备的。为了不调大家胃口，开始剧透：这个汇编函数就是 switch_to，它是由 C 语言函数 schedule 来调用的，因此为了不破坏主调函数 schedule 的寄存器，咱们要在汇编代码中保存这 5 个寄存器，保存的位置就是这个线程栈。这内容有点超前了，到以后介绍 switch_to 的实现时您就清楚了。当然，如果您想图省事的话，也可以保存所有的 32 位通用寄存器，如用指令 pushad。

esp 的值会由调用约定来保证，因此我们不打算保护 esp 的值。在我们的实现中，由被调函数保存除 esp 外的 4 个寄存器，这就是线程栈 thread_stack 前 4 个成员的作用，我们将来用 switch_to 函数切换时，先在线程栈 thread_stack 中压入这 4 个寄存器的值。

这块内容光凭局部描述还是无法彻底说清楚，还是结合代码 9-4 从全局上体会比较好，最好是跟一下代码从整体上搞清楚。咱们先继续看下面的内容。

第 64 行有一句注释：“以下仅供第一次被调度上 CPU 时使用”，这指的是下面的三行内容：

```
void (*unused_retaddr);
thread_func* function;
void* func_arg;
```

其中，unused_retaddr 用来充当返回地址，在返回地址所在的栈帧占个位置，因此 unused_retaddr 中的值并不重要，仅仅起到占位的作用。

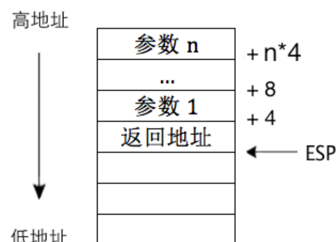
function 是由函数 kernel_thread 所调用的函数名（Kernel_thread 在 thread.c 中有介绍，咱们一会儿说），即 function 是在线程中执行的函数。

func_arg 是由 kernel_thread 所调用的函数所需的参数，即 function 的参数，因此最终的情形是：在线程中调用的是 function(func_arg)。

欲知详情，咱们先回顾下函数调用时发生的情况（您已进入属于 ABI 的范畴^^）。

函数在执行前，如果该函数有参数的话，调用者一定会按照调用约定，先把参数压到栈中。

在 C 语言层面，函数的执行都是由调用者发起调用的，这通过 call 指令完成，此指令会在栈中留下返回地址。因此被调用的函数在执行时，会认为调用者已经把返回地址留在栈中，而且是在栈顶的位置。也就是说当进入到被调用函数中执行时，栈中的情形应该如图 9-9 所示。



▲图 9-9　栈帧情况

为了解释清楚，这里还是要剧透一下，将来我们这里的被调用者是 eip 所指向的 kernel_thread 函数，当 kernel_thread 开始执行时，处理器会认为当前栈顶“应该是”调用者的返回地址，因此它会从当前栈顶+4 的位置找参数。

我们在线程中待执行的函数 function 及其参数 func_arg 是由 kernel_thread 去调用执行的，它们两个作为 kernel_thread 的参数，形如这样的形式：

```
kernel_thread(thread_func* func, void* func_arg) {
func(func_arg);
}
```

进入到函数 kernel_thread 时，栈顶处是返回地址，因此栈顶+4 的位置保存的是 function，栈顶+8 保存的是 func_arg。

x86 处理器被程序计数器 CS 和 EIP“牵着鼻子走”，这两个寄存器中的值才是它下一条要执行的指令地址，因此，执行流存在的原因就是程序中包含改变 CS 或 EIP 的值的指令，这些指令有 call、jmp、ret 等（注意，可不是 mov 哦）。

大伙儿都已经知道，call 指令属于“有去有回”的指令，它在“去”之前先在栈中（进入被调函数时的栈顶处）留下返回地址，它的“回”则需要在 ret 指令的配合下才能完成，ret 将栈顶的值当作 call 留下的返回地址，在保证栈顶值正确的情况下，ret 能把处理器重新带回到主调函数中。这里我们灵活运用了 ret 指令，即在没有 call 指令的前提下，直接在栈顶装入函数的返回地址，再利用 ret 指令执行该函数，当然这是在内联汇编下做的。

我们说下具体的例子，在一会介绍 thread.c 时大伙儿就会知道，kernel_thread 函数并不是通过调用 call 指令的形式执行的，而是咱们用汇编指令 ret“返回”执行的，也就是函数 kernel_thread 作为“某个函数”（此函数暂时为 thread_start）的返回地址，通过 ret 指令使函数 kernel_thread 的偏移地址（段基址为 0）被载入到处理器的 EIP 寄存器，从而处理器开始执行函数 kernel_thread，但由于它的执行并不是通过 call 指令实现的，所以进入到函数 kernel_thread 中执行时，栈中并没有返回地址。

也许您对为什么要用 ret 指令执行 kernel_thread 函数还是有些“耿耿于怀”，哈哈，理解，小弟在这解释一下。大伙儿已经知道线程栈 struct thread_stack 有两个作用。

第 1 个作用是在线程首次运行时，线程栈用于存储创建线程所需的相关数据。和线程有关的数据应该都在该线程的 PCB 中，这样便于线程管理，避免为它们再单独维护数据空间。创建线程之初，要指定在线程中运行的函数及参数，因此，把它们放在位于 PCB 所在页的高地址处的 0 级栈中比较合适，该处就是线程栈的所在地址。

第 2 个作用是用在任务切换函数 switch_to 中的，这是线程已经处于正常运行后线程栈所体现的作用。为解释清楚这个疑惑，现在不得不告诉大家 switch_to 函数是我们用汇编语言实现的，它是被内核调度器函数调用的，因此这里面涉及到主调函数寄存器的保护，就是 ebp、ebx、edi 和 esi 这 4 个寄存器，前面那段英文已经阐述了，它们属于主调函数（这里是指调度器函数），咱们要在被调用函数 switch_to 中将它们保护起来，也就是将它们保存在栈中，这必然涉及到压栈指令 push，用单纯的汇编语言比 C 语言内嵌汇编的方式要方便一些，请大伙儿见谅。

您看，switch_to 既然是汇编程序，从其返回时，必然要用到 ret 指令，因此为了同时满足这 2 个作用，或者说是为了让作用 1“配合”作用 2，我们最初先在线程栈中装入适合的返回地址及参数，使作用 2 中 switch_to 的 ret 指令也满足创建线程时的作用 1。

好啦，在不了解全局的情况下对此也就解释这么多了，在后面看实现部分时您也许会一下子全明白了。

函数调用在 C 语言中只有高级语言的调用形式，即 function(args)，在 C 语言层面，函数的执行通常被编译为 call 指令调用的形式，很少有通过 ret 来执行函数的，因此在某个函数中执行时，按照规定，栈顶应该是某个主调函数通过 call 指令调用该函数时留下的返回地址，在栈顶之上（高地址）的栈帧中是被调函数的参数，这才是按照正常的套路出牌。也就是说，栈中参数的位置是以栈顶为基准的，如图 9-9 所示，尽管我们不是用 call 指令调用函数，但依然必须按照这个规则办事。

为了满足 C 语言的调用形式，使 kernel_thread 以为自己是通过“正常渠道”，也就是 call 指令调用执行的，当前栈顶必须得是返回地址，故参数 unused_ret 只为占位置充数，由它充当栈顶，其值充当返回地址，所以它的值是多少都没关系，因为咱们将来不需要通过此返回地址“返回”，咱们的目的是让 kernel_thread 去调用 func(func_arg)，也就是“只管继续向前执行”就好了，此时不需要“回头”。总之我们只要保留这个栈帧位置就够了，为的是让函数 kernel_thread 以为栈顶是它自己的返回地址，这样便有了一个正确的基准，并能够从栈顶+4 和栈顶+8 的位置找到参数 func 和 func_arg。否则，若没有占位成员 unused_ret 的话，处理器依然把栈顶当作返回地址作为基准，以栈顶向上+4 和+8 的地方找参数 func 和 func_arg，但由于没有返回地址，此时栈顶就是参数 func，栈顶+4 就是 func_arg，栈顶+8 的值目前未知，要看实际编译情况，因此处理器便找错了栈帧位置，后果必然出错。

注意，这里所说的“只管继续向前执行”，只是函数第一次在线程中执行的情况，即前面所说的栈 thread_stack 的第 1 个作用。在第 2 个作用中，会由调度函数 switch_to 为其留下返回地址，这时才需要返回。

再说明一下 func_arg 的类型 void*，其是无类型指针，目的是为表示所有参数类型，在 kernel_thread 调用这些函数时，用 void*来通用表示参数，在 kernel_thread 中被调用的函数 function，它知道自己需要什么类型的参数，因此在 function 函数体中可以自己转换成想要的类型。

代码 9-1-2 中的结构体 struct task_struct 是定义的 PCB，这是最最简单的 PCB 结构了，现在先从小做起，以后再丰富它吧。其中 self_kstack 是各线程的内核栈顶指针，当线程被创建时，self_kstack 被初始化为自己 PCB 所在页的顶端。之后在运行时，在被换下处理器前，我们会把线程的上下文信息（也就是寄存器映像）保存在 0 特权级栈中，self_kstack 便用来记录 0 特权级栈在保存线程上下文后的新的栈顶，在下一次此线程又被调度到处理器上时，可以把 self_kstack 的值加载到 esp 寄存器，这样便从 0 特权级栈中获取了线程上下文，从而可以加载到处理器中运行。

status 用于记录线程状态，其类型便是前面定义的枚举结构 enum task_status。

priority 用于记录线程优先级，进程或线程都要有个优先级，此优先级咱们用来决定进程或线程的时间片，即被调度到处理器上的运行时间。

name[16]用于记录任务（线程或进程）的名字，长度是 16，即任务名最长不过 16 个字符。

stack_magic 是栈的边界标记，用于检测栈的溢出。咱们 PCB 和 0 级栈是在同一个页中，栈位于页的顶端并向下发展，因此担心压栈过程中会把 PCB 中的信息给覆盖，所以每次在线程或进程调度时要判断是否触及到了进程信息的边界，也就是判断 stack_magic 的值是否为初始化的内容，stack_magic 实际上就是个魔数。

最后说明一下，中断栈 intr_stack 和线程栈 thread_stack 都位于线程的内核栈中，也就是都位于 PCB 的高地址处。

9.2.2 线程的实现

上一节中咱们花了较大篇幅介绍了栈相关的基础数据结构，但说实话只介绍这些局部的话，很难让大伙儿把整个原理搞清楚，必须得结合其他使用这些结构的代码，让您知道它们是如何配合使用的才行，这就是本节要做的事。请看代码 9-2。

代码 9-2　（project/c9/a/thread/thread.c）

```
 1 #include "thread.h"
 2 #include "stdint.h"
 3 #include "string.h"
 4 #include "global.h"
 5 #include "memory.h"
 6
 7 #define PG_SIZE 4096
 8
 9 /* 由 kernel_thread 去执行 function(func_arg) */
10 static void kernel_thread(thread_func* function, void* func_arg) {
11    function(func_arg);
12 }
13
14 /* 初始化线程栈 thread_stack,
   将待执行的函数和参数放到 thread_stack 中相应的位置 */
15 void thread_create(struct task_struct* pthread, thread_func function, void* func_arg) {
16    /* 先预留中断使用栈的空间，可见 thread.h 中定义的结构 */
17    pthread->self_kstack -= sizeof(struct intr_stack);
18
19    /* 再留出线程栈空间，可见 thread.h 中定义 */
20    pthread->self_kstack -= sizeof(struct thread_stack);
21    struct thread_stack* kthread_stack = (struct thread_stack*)pthread->self_kstack;
22    kthread_stack->eip = kernel_thread;
23    kthread_stack->function = function;
24    kthread_stack->func_arg = func_arg;
25    kthread_stack->ebp = kthread_stack->ebx = \
      kthread_stack->esi = kthread_stack->edi = 0;
26 }
27
28 /* 初始化线程基本信息 */
29 void init_thread(struct task_struct* pthread, char* name, int prio) {
30    memset(pthread, 0, sizeof(*pthread));
31    strcpy(pthread->name, name);
32    pthread->status = TASK_RUNNING;
33    pthread->priority = prio;
34 /* self_kstack 是线程自己在内核态下使用的栈顶地址 */
35    pthread->self_kstack = (uint32_t*)((uint32_t)pthread + PG_SIZE);
36    pthread->stack_magic = 0x19870916;    // 自定义的魔数
37 }
38
39 /* 创建一优先级为 prio 的线程，线程名为 name,
   线程所执行的函数是 function(func_arg) */
40 struct task_struct* thread_start(char* name, \
                                    int prio, \
                                    thread_func function, \
                                    void* func_arg) {
41 /* pcb 都位于内核空间，包括用户进程的 pcb 也是在内核空间 */
42    struct task_struct* thread = get_kernel_pages(1);
43
44    init_thread(thread, name, prio);
45    thread_create(thread, function, func_arg);
46
47    asm volatile ("movl %0, %%esp; \
pop %%ebp; pop %%ebx; pop %%edi; pop %%esi; \
ret": : "g" (thread->self_kstack) : "memory");
48    return thread;
49 }
```

本节仅仅是先让线程跑起来，因此代码 9-2 中内容比较少，对目前来说够用了。

咱们先从最下面的函数 thread_start 说起，此函数接受 4 个参数，name 为线程名，prio 为线程的优先级，要执行的函数是 function，func_arg 是函数 function 的参数。thread_start 的功能是创建一优先级为 prio 的线程，线程名为 name，线程所执行的函数是 function(func_arg)。

在函数体内，先通过 get_kernel_pages(1)在内核空间中申请一页内存，即 4096 字节，将其赋值给新创建的 PCB 指针 thread，即 struct task_struct* thread。注意，由于 get_kernel_page 返回的是页的起始地址，故 thread 指向的是 PCB 的最低地址。

请大伙注意，无论是进程或线程的 PCB，这都是给内核调度器使用的结构，属于内核管理的数据，因此将来用户进程的 PCB 也依然要从内核物理内存池中申请。

接下来调用 init_thread（thread, name, prio）来初始化刚刚创建的 thread 线程。此函数定义在第 29 行，它接受 3 个参数，pthread 是待初始化线程的指针，name 是线程名称，prio 是线程的优先级，此函数功能是将 3 个参数写入线程的 PCB，并且完成 PCB 一级的其他初始化。

在 init_thread 中，先调用 memset(pthread, 0, sizeof(*pthread))将 pthread 所在的 PCB 清 0，即清 0 一页。

再通过 strcpy(pthread->name, name)将线程名写入 PCB 中的 name 数组中。

接下来为线程的状态 pthread->status 赋值，由于目前仅仅为了演示，故直接将 status 置为 TASK_RUNNING，以后再按照正常的逻辑为状态赋值。

接下来再将 prio 赋值给 pthread->priority，目前的优先级没什么用，将来它的作用体现任务（线程和进程的统称）在处理器上执行的时间片长度，即优先级越高，执行的时间片越长。

pthread->self_kstack 是线程自己在 0 特权级下所用的栈，在线程创建之初，它被初始化为线程 PCB 的最顶端，即(uint32_t)pthread + PG_SIZE。

PCB 的上端是 0 特权级栈，将来线程在内核态下的任何栈操作都是用此 PCB 中的栈，如果出现了某些异常导致入栈操作过多，这会破坏 PCB 低处的线程信息。为此，需要检测这些线程信息是否被破坏了，stack_magic 被安排在线程信息的最边缘，作为它与栈的边缘。目前用不到此值，以后在线程调度时会检测它。pthread->stack_magic 自定义个值就行，我这里用的是 0x19870916，这与代码功能无关。

回来继续看 thread_start 函数。在调用完 init_thread 后，它又调用了 thread_create 创建了线程。

thread_create 接受 3 个参数，pthread 是待创建的线程的指针，function 是在线程中运行的函数，func_arg 是 function 的参数。函数的功能是初始化线程栈 thread_stack，将待执行的函数和参数放到 thread_stack 中相应的位置。

在 thread_create 中，pthread->self_kstack -= sizeof（struct intr_stack）是为了预留线程所使用的中断栈 struct intr_stack 的空间，这有两个目的。

（1）将来线程进入中断后，位于 kernel.S 中的中断代码会通过此栈来保存上下文。

（2）将来实现用户进程时，会将用户进程的初始信息放在中断栈中。

因此，必须要事先把 struct intr_stack 的空间留出来。

pthread->self_kstack 在 init_thread 中已经被指向了 PCB 的最顶端，所以现在要减去中断栈的大小。此时 pthread->self_kstack 指向 PCB 中的中断栈下面的地址。

在下一行中，struct thread_stack* kthread_stack 定义了线程栈指针，这个就是上一节中我们介绍的占位成员 unused_retaddr 所在的栈。

对比上一节中介绍的线程栈 struct thread_stack 的结构，我们看下这三行赋值：

```
kthread_stack->eip = kernel_thread;
kthread_stack->function = function;
kthread_stack->func_arg = func_arg;
```

其中的 function 就是函数 thread_start 的形参 function 所指向的函数，其中的 func_arg 就是 thread_start 的形参 func_arg 的值，这三行就是为能够在 kernel_thread 中调用 function(func_arg)做准备。

正如我们上一节中说过，eip 指向 kernel_thread，它定义在代码 9-2 的最上面第 10 行处，大伙儿先移步看看它的实现。

kernel_thread 接受两个参数，function 是 kernel_thread 中调用的函数，func_arg 是 function 的参数，因此 kernel_thread 函数的功能就是调用 function(func_arg)。

上一节中说过了，kernel_thread 并不是通过 call 指令调用的，而是通过 ret 来执行的（一会儿我们继续介绍 thread_start 时您就知道 ret 在哪里了），因此无法按照正常的函数调用形式传递 kernel_thread 所需要的参数，如这样调用是不行的：kernel_thread(function, func_arg)，只能将参数放在 kernel_thread 所用的栈中，即处理器进入 kernel_thread 函数体时，栈顶为返回地址，栈顶+4 为参数 function，栈顶+8 为参数

func_arg，这就是上面三行中标有下画线的两行代码的作用。

接下来把 ebp，ebx，esi，edi 这 4 个寄存器初始化为 0，因为线程中的函数尚未执行，在执行过程中寄存器才会有值，此时置为 0 即可。

另外说一下 kthread_stack->unused_retaddr 是不需要赋值的，就是用来占位的，因此我们代码中并没有对它处理。

thread_create 就介绍完了，咱们回来继续看 thread_start。

目前只差最后的汇编指令没说了。汇编指令在第 47 行，这是我们为演示线程运行的临时方案，以后就没这么“简陋”了。此汇编代码是开启线程的钥匙，不过这个钥匙还是蛮长的，哈哈，咱们逐句分析。

在输出部分，"g" (thread->self_kstack)使 thread->self_kstack 的值作为输入，采用通用约束 g，即内存或寄存器都可以。

在汇编语句部分，movl %0, %%esp，也就是使 thread->self_kstack 的值作为栈顶，此时 thread->self_kstack 指向线程栈的最低处，这是我们在函数 thread_create 中设定的。

接下来的这连续 4 个弹栈操作：pop %%ebp; pop %%ebx; pop %%edi; pop %%esi 使之前初始化的 0 弹入到相应寄存器中。

到了关键时刻了，我们马上要执行 ret 了，我们知道，ret 会把栈顶的数据作为返回地址送上处理器的 EIP 寄存器。

回忆下此时栈顶的数据是什么，我想您早已经知道了，就是在 thread_create 中为 kthread_stack->eip 所赋的值——kernel_thread。因此，在执行 ret 后，处理器会去执行 kernel_thread 函数。接着在 kernel_thread 函数中会调用传给函数 function(func_arg)。

在执行完这句汇编后，线程就会开始执行，好啦，代码 9-2 介绍完了。接下来得找个地方调用 thread_start，这是我们创建线程的入口。您猜到了，还是在主函数 main 中。请见代码 9-3。

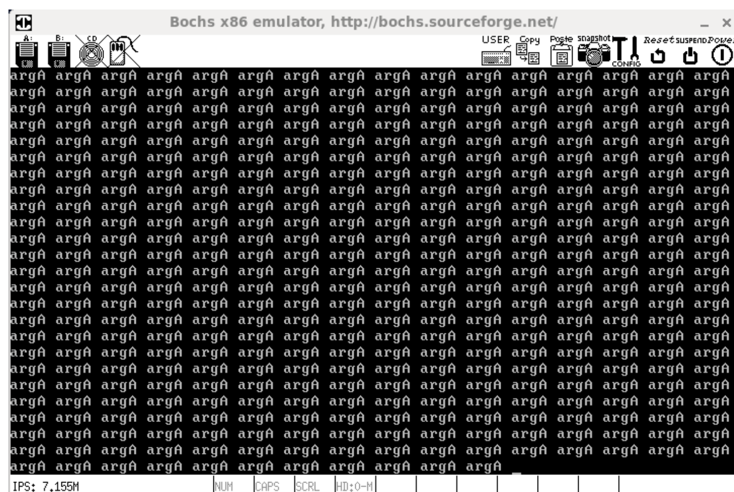
代码 9-3　（project/c9/a/kernel/main.c）

```
 1 #include "print.h"
 2 #include "init.h"
 3 #include "thread.h"
 4
 5 void k_thread_a(void*);
 6
 7 int main(void) {
 8     put_str("I am kernel\n");
 9     init_all();
10
11     thread_start("k_thread_a", 31, k_thread_a, "argA ");
12
13     while(1);
14     return 0;
15 }
16
17 /* 在线程中运行的函数 */
18 void k_thread_a(void* arg) {
19 /* 用 void*来通用表示参数，
   被调用的函数知道自己需要什么类型的参数，自己转换再用 */
20     char* para = arg;
21     while(1) {
22         put_str(para);
23     }
24 }
```

代码 9-3 中，我们在头文件中加入了 thread.h，并且在第 11 行中调用 thread_start("k_thread_a", 31, k_thread_a, "argA ")创建了新线程。线程名字为 k_thread_a，优先级为 31（此时没什么用，先留着），此线程运行的函数是 k_thread_a，它定义在第 18 行，功能就是打印参数 arg。我们传给 thread_start 的第 4 个参数是字符串“argA”，因此线程在运行时会在屏幕上循环输出 arg_A。

编译运行，结果如图 9-10 所示。

如图，满屏的 argA，这说明我们暂时成功了。线程初战告捷，本节工作暂告一段落，下节再见。



▲图 9-10 线程运行结果

9.3 核心数据结构，双向链表

程序=算法+数据结构。

说来惭愧，把这么著名的一句话写在本节开头，我还是有些许脸红的。限于咱们为了简单以及本人的能力实在有限，到目前为止我们并没有采用什么高深的算法。

不过算法再怎么简单，数据结构还是必不可少的，数据需要存储到合适的数据结构中才能获得更高效的管理。就拿队列来说，它是一种先入先出的数据结构，很多需要保证时序的数据一般都用队列来存储，实现队列的方式有很多，最简单的就是用数组，稍微复杂一些可以用链表。

在咱们的内核中也要用到队列，比如进程的就绪队列、锁的等待队列等，为了维护内核中的各种队列，咱们本节要实现自己的链表——双向链表。

数据结构课程我估计大家都学过，我看还是直接上代码吧，由于队列属于内核的数据结构，故我们在 lib/kernel/下创建 list.h 及 list.c，我们先看看头文件的定义，请见代码 9-4。

代码 9-4 （project/c9/b/lib/kernel/list.h）

```
 1 #ifndef __LIB_KERNEL_LIST_H
 2 #define __LIB_KERNEL_LIST_H
 3 #include "global.h"
 4
 5 #define offset(struct_type,member) (int)(&((struct_type*)0)->member)
 6 #define elem2entry(struct_type, struct_member_name, elem_ptr) \
 7            (struct_type*)((int)elem_ptr - offset(struct_type, struct_member_name))
 8
 9 /**********   定义链表结点成员结构   ***********
10 *结点中不需要数据成元，只要求前驱和后继结点指针*/
11 struct list_elem {
12     struct list_elem* prev; // 前躯结点
13     struct list_elem* next; // 后继结点
14 };
15
16 /* 链表结构，用来实现队列 */
17 struct list {
18 /* head 是队首，是固定不变的，不是第 1 个元素，第 1 个元素为 head.next */
19     struct list_elem head;
20 /* tail 是队尾，同样是固定不变的 */
21     struct list_elem tail;
22 };
23
24 /* 自定义函数类型 function，用于在 list_traversal 中做回调函数 */
```

```
25 typedef bool (function)(struct list_elem*, int arg);
26
27 void list_init (struct list*);
28 void list_insert_before(struct list_elem* before, struct list_elem* elem);
29 void list_push(struct list* plist, struct list_elem* elem);
30 void list_iterate(struct list* plist);
31 void list_append(struct list* plist, struct list_elem* elem);
32 void list_remove(struct list_elem* pelem);
33 struct list_elem* list_pop(struct list* plist);
34 bool list_empty(struct list* plist);
35 uint32_t list_len(struct list* plist);
36 struct list_elem* list_traversal(struct list* plist, function func, int arg);
37 bool elem_find(struct list* plist, struct list_elem* obj_elem);
38 #endif
```

第 11 行定义的是结构体 struct list_elem，它是链表中结点的结构，这是链表的核心。一般的链表结点中除了前驱或后继结点的指针外，还包括数据成员，即链表结点是数据的存储单元。您看到了，本结点只有前驱结点指针 prev 和后继结点指针 next，不包含数据成员，结点中为什么没有数据成员呢？哈哈，因为不需要。

存储数据只是链表的部分功能，它最主要的功能是“链”，咱们的链表单纯是为了将已有的数据以一定的时序链起来，因此不是为了存储，所以结点中不需要数据成员。

第 16～22 行是定义双向链表的结构体，链表都有访问的起始入口，我们这里定义首尾两个入口，用 head 表示链表开头，tail 表示链表结尾，这样链表就可以按照从前到后或从后到前的顺序来遍历了。

head 和 tail 这两个成员是固定不变的，它们是链表固定的两个入口。新插入的结点不会替代它们的位置，只是会插入在 head 和 tail 之间。就像火车一样，两头的火车头是不变的，要增加车厢，只是往中间加。

你看，head 和 tail 的数据类型都是 struct list_elem，head.next 是链表中第 1 个元素结点，tail.prev 是链表中最后一个元素结点，以后在链表中插入结点时，都是插入在 head 和 tail 之间。至于 head.prev 和 tail.next，它们的值无意义，在初始化时会被置为空。

文件开头定义的两个宏 elem2entry 和 offset，这是为了将结点元素转换成实际元素项，将来用到时再讨论介绍啦。

list.c 中用到了关中断的函数 intr_disable，在介绍 list.c 之前，有必要先和大伙儿交待一下为什么这样做。

系统中有些数据是公共资源，对于它的修改应该保证是原子操作。学过操作系统的同学都知道有个临界区的概念，简单来说，访问公共资源的程序片段叫临界区，临界区通常是指在不同线程中的、修改同一公共资源的指令区域。临界区中的代码应该属于原子操作，要么不执行，要么就全部执行完（就像数据库中的事务一样），说白了就是怕某线程临界区中的代码未全部执行完就被换下处理器，然后另一个线程的临界区代码又对此公共资源有读写，于是造成公共资源数据的错误，这就是资源竞争的问题。即使现在不清楚这些概念也没关系，过些天咱们介绍“锁”的时候会和大家细说。

咱们刚学过位图，我还是用位图的操作给大伙举个例子吧。

比如当内核线程 A 在位图中找到空闲位（bit 值为 0）时，还没来得及将其分配出去（也就是将位图中该 bit 的值改为 1），这时候运行的时间片到了，被换下了 CPU。将线程 B 换上 CPU 运行，线程 B 也要扫描位图，同样也找到了线程 A 当初发现的空闲位，于是将该位的值置为 1，表示已分配。当线程 A 又被换上 CPU 进行的时候，由于它不知道曾经找到的空闲位已经被线程 B 抢走了，所以它的下一步工作是将该空闲位置 1（已经由线程 B 置过 1 了），然后重复使用已经分配给线程 B 的资源，于是引起了冲突。

咱们将来的进程调度机制依靠时钟中断，此处把中断关闭，就避免了在检索位图时被换下 CPU 的可能。所以，对于此例子中的位图操作，一定要保证是在关中断的情况下进行。

有没有同学疑惑：哎？那咱们之前的位图操作也没关中断啊，这里不是要求关中断吗？是这样的，实现原子操作，关中断只是一种方式，将来咱们会用锁来保证。

好，我就当大伙儿已经理解使用 intr_disable 函数关中断的意义了，那咱们下面看下 list.c，见代码 9-5。

代码 9-5　（project/c9/b/lib/kernel/list.c）

```
1 #include "list.h"
2 #include "interrupt.h"
```

```
 3
 4 /* 初始化双向链表 list */
 5 void list_init (struct list* list) {
 6    list->head.prev = NULL;
 7    list->head.next = &list->tail;
 8    list->tail.prev = &list->head;
 9    list->tail.next = NULL;
10 }
11
12 /* 把链表元素 elem 插入在元素 before 之前 */
13 void list_insert_before(struct list_elem* before, struct list_elem* elem) {
14    enum intr_status old_status = intr_disable();
15
16 /* 将 before 前驱元素的后继元素更新为 elem，暂时使 before 脱离链表*/
17    before->prev->next = elem;
18
19 /* 更新 elem 自己的前驱结点为 before 的前驱，
20  * 更新 elem 自己的后继结点为 before，于是 before 又回到链表 */
21    elem->prev = before->prev;
22    elem->next = before;
23
24 /* 更新 before 的前驱结点为 elem */
25    before->prev = elem;
26
27    intr_set_status(old_status);
28 }
29
30 /* 添加元素到列表队首，类似栈 push 操作 */
31 void list_push(struct list* plist, struct list_elem* elem) {
32    list_insert_before(plist->head.next, elem); // 在队头插入 elem
33 }
34
35 /* 追加元素到链表队尾，类似队列的先进先出操作 */
36 void list_append(struct list* plist, struct list_elem* elem) {
37    list_insert_before(&plist->tail, elem);    // 在队尾的前面插入
38 }
39
40 /* 使元素 pelem 脱离链表 */
41 void list_remove(struct list_elem* pelem) {
42    enum intr_status old_status = intr_disable();
43
44    pelem->prev->next = pelem->next;
45    pelem->next->prev = pelem->prev;
46
47    intr_set_status(old_status);
48 }
49
50 /* 将链表第一个元素弹出并返回，类似栈的 pop 操作 */
51 struct list_elem* list_pop(struct list* plist) {
52    struct list_elem* elem = plist->head.next;
53    list_remove(elem);
54    return elem;
55 }
56
57 /* 从链表中查找元素 obj_elem，成功时返回 true，失败时返回 false */
58 bool elem_find(struct list* plist, struct list_elem* obj_elem) {
59    struct list_elem* elem = plist->head.next;
60    while (elem != &plist->tail) {
61       if (elem == obj_elem) {
62           return true;
63       }
64       elem = elem->next;
65    }
66    return false;
67 }
68
69 /* 把列表 plist 中的每个元素 elem 和 arg 传给回调函数 func，
70  * arg 给 func 用来判断 elem 是否符合条件.
71  * 本函数的功能是遍历列表内所有元素，逐个判断是否有符合条件的元素。
72  * 找到符合条件的元素返回元素指针，否则返回 NULL */
73 struct list_elem* list_traversal(struct list* plist, function func, int arg) {
74    struct list_elem* elem = plist->head.next;
```

```
75  /* 如果队列为空，就必然没有符合条件的结点，故直接返回 NULL */
76     if (list_empty(plist)) {
77          return NULL;
78     }
79
80     while (elem != &plist->tail) {
81         if (func(elem, arg)) {
// func 返回 ture，则认为该元素在回调函数中符合条件，命中，故停止继续遍历
82             return elem;
83         }      // 若回调函数 func 返回 true，则继续遍历
84         elem = elem->next;
85     }
86     return NULL;
87 }
88
89 /* 返回链表长度 */
90 uint32_t list_len(struct list* plist) {
91      struct list_elem* elem = plist->head.next;
92      uint32_t length = 0;
93      while (elem != &plist->tail) {
94         length++;
95         elem = elem->next;
96      }
97      return length;
98 }
99
100 /* 判断链表是否为空，空时返回 true，否则返回 false */
101 bool list_empty(struct list* plist) {         // 判断队列是否为空
102     return (plist->head.next == &plist->tail ? true : false);
103 }
```

咱们从上往下把函数逐个介绍下。

函数 list_init 只接受一个参数 list，功能是初始化双向链表 list。此时钟表是空的，因此函数内部的初始化工作就是把表头 head 和表尾 tail 连接起来，即“list->head.next = &list->tail”和“list->tail.prev = &list->head”。head.prev 和 tail.next 的值无意义，因此被置为 NULL。

函数 list_insert_before 接受两个参数，before 和 elem，它们皆为链表结点的指针，此函数功能是把链表元素 elem 插入在元素 before 之前。

由于队列是公共资源，对于它的修改一定要保证为原子操作，所以在函数体的第 14 行通过 intr_disable 将中断关闭，旧中断状态用变量 old_status 保存，以此保证下面的 4 个操作的原子性（不可拆分、连续性），操作结束后在第 27 行通过“intr_set_status(old_status)”将中断恢复。

第 17 行的“before->prev->next = elem”，其中 before->prev 是获取 before 的前驱元素，之后再用->next 获取此前躯元素的下个结点，此时将其赋值为 elem，因此本行代码的功能是将 before 前驱元素的后继元素更新为 elem，暂时使 before 脱离链表。

由于这是双向链表，第 17 行只是完成了从前到后单向的更新，还要保证从后往前也能链接上才行。因此在第 21 行，将 elem 的前驱结点更新为 before 的前驱，即代码“elem->prev = before->prev”。由于是将 elem 插入在 before 之前，故需要将 elem 的后继结点更新为 before，因此在第 22 行，通过代码“elem->next = before”完成。此时 elem 已经替代了 before 在链表中的位置。

接下来，要保证从后往前能访问到 elem，由于 before 已经在 elem 的后面，故需要将 before 的前驱结点指向 elem，于是在第 25 行用代码“before->prev = elem”完成。list_insert_before 就是这么简单。

函数 list_push 接受两个参数，plist 是链表，elem 是链表结点，功能是添加元素 elem 到列表 plist 的队首，其实这就是栈的特性，后进先出，因此相当于用链表实现了栈。其内部是调用“list_insert_before(plist->head.next, elem)”实现的，即在队头 head.next 的前面插入 elem。

函数 list_append 接受两个参数，plist 是链表，elem 是链表结点，功能是添加元素 elem 到列表 plist 的队尾，其实这就是队列的特性，先进先出，因此相当于用链表实现了线性队列。其内部是调用“list_insert_before(&plist->tail, elem)”实现的，就是在队尾 tail 的前面插入 elem。

函数 list_remove 接受一个参数，链表结点 pelem，功能是将 pelem 从链表中去除。原理就是让 pelem

前驱的后继结点，指向 pelem 的后继结点，即“pelem->prev->next = pelem->next”，让 pelem 后继的前驱结点指向 pelem 的前驱结点，即“pelem->next->prev = pelem->prev”。此函数也是对链表修改，所以咱们为了保证原子性，同样先将中断关闭，完成操作后再恢复。

函数 list_pop 只接受一个参数，链表 plist，功能是将链表 plist 的第一个结点弹出，类似出栈操作。内部实现就是先获取链表第一个元素 plist->head.next，将其存到指针 elem 中，即“struct list_elem* elem = plist->head.next”，然后再调用“list_remove(elem)”将其从链表中删除，随后再通过“return elem”将其返回。

函数 elem_find 接受两个参数，链表 plist 和待查找的结点 obj_elem，功能是从链表 plist 中查找元素 obj_elem，成功时返回 true，失败时返回 false。实现原理是把链表的第一个结点作为入口，通过 while 循环遍历链表中所有结点，如果找到就返回 true，否则遍历完整个链表后都没有找到的话就返回 false。

函数 list_traversal 接受三个参数，链表 plist、回调函数 func 及回调函数的参数 arg，功能是遍历列表内所有元素，逐个判断是否有符合“条件”的元素结点，找到符合条件的结点返回结点指针，否则返回 NULL。其中的“条件”是由回调函数 func 来判断的，如果条件成立，func(arg)会返回 true，否则会返回 false（list_traversal 有些类似 ruby 中的枚举用法）。内部实现也是遍历所有结点，用变量 elem 保存每一个结点，对各个结点都调用“func(elem, arg)”，如果 func 返回 true，则表示找到了目标结点，不再继续遍历，通过 return elem 返回找到的结点指针。否则整个链表遍历结束也没有找到符合条件的结点时就返回 NULL。

函数 list_len 只接受一个参数，链表 plist，功能是返回链表长度，即链表中结点的个数。实现原理也是通过循环遍历所有结点，用变量 length 计数，最后再将计数返回，不再细说。

函数 list_empty 只接受一个参数，链表 plist，功能是判断链表 plist 是否为空，空时返回 true，否则返回 false。其内部实现较直接，就一句代码，即“return (plist->head.next == &plist->tail ? true : false)”，原理是判断链表 plist 第一个结点是否指向链表 plist 的尾。

好啦，list.c 就介绍完了，由于链表只是基础组件，目前还未派上用场，所以本节无实验可做，大伙就早点休息啦。

9.4 多线程调度

之前咱们已经完成了单个线程的执行，那只是咱们小试身手的第一步，本节咱们要完成真正意义上的多线程，让多个线程在调度器的调度下轮流执行。

9.4.1 简单优先级调度的基础

本节任务是把 thread.c 和 thread.h 进一步完善，在原有线程的基础上添加新的功能，咱们的目标是完成线程的轮询调度。

为实现这一目标，这里有一些基础工作要先完成，咱们先看看 thread.h 又增加的内容，请见代码 9-6。

代码 9-6 （project/c9/c/thread/thread.h）

```
…略
73 /* 进程或线程的 pcb，程序控制块 */
74 struct task_struct {
75     uint32_t* self_kstack;        // 各内核线程都用自己的内核栈
76     enum task_status status;
77     char name[16];
78     uint8_t priority;
79     uint8_t ticks;                // 每次在处理器上执行的时间嘀嗒数
80
81 /* 此任务自上 cpu 运行后至今占用了多少 cpu 嘀嗒数，
82  也就是此任务执行了多久*/
83     uint32_t elapsed_ticks;
84
85 /* general_tag 的作用是用于线程在一般的队列中的结点 */
86     struct list_elem general_tag;
87
88 /* all_list_tag 的作用是用于线程队列 thread_all_list 中的结点 */
```

```
89     struct list_elem all_list_tag;
90
91     uint32_t* pgdir;              // 进程自己页表的虚拟地址
92     uint32_t stack_magic;         // 用这串数字做栈的边界标记
                                   // 用于检测栈的溢出
93 };
以下是函数声明……
```

struct task_struct 结构中新增了几个成员。

ticks 之前咱们也添加过了，当时限于演示，没有给出其实际用途。它是任务每次被调度到处理器上执行的时间嘀嗒数，也就是我们所说的任务的时间片，每次时钟中断都会将当前任务的 ticks 减 1，当减到 0 时就被换下处理器。

ticks 和上面的 priority 要配合使用。priority 表示任务的优先级，咱们这里优先级体现在任务执行的时间片上，即优先级越高，每次任务被调度上处理器后执行的时间片就越长。当 ticks 递减为 0 时，就要被时间中断处理程序和调度器换下处理器，调度器把 priority 重新赋值给 ticks，这样当此线程下一次又被调度时，将再次在处理器上运行 ticks 个时间片。

elapsed_ticks 用于记录任务在处理器上运行的时钟嘀嗒数，从开始执行，到运行结束所经历的总时钟数。

general_tag 的类型是 struct list_elem，也就是 general_tag 是双向链表中的结点。它是线程的标签，当线程被加入到就绪队列 thread_ready_list 或其他等待队列中时，就把该线程 PCB 中 general_tag 的地址加入队列。

一个 struct list_elem 类型的结点只有一对前驱和后继指针，它只能被加入一个队列，但在咱们的系统中，为管理所有线程，还存在一个全部线程队列 thread_all_list，因此线程还需要另外一个标签，即 all_list_tag。

all_list_tag 的类型也是 struct list_elem，它专用于线程被加入全部线程队列时使用。

也许大伙儿可能有疑惑，这两个标签又不是线程，加入队列后就能用于某种管理了？此处先提一下，这两个标签仅仅是加入队列时用的，将来从队列中把它们取出来时，还需要再通过 offset 宏与 elem2entry 宏的“反操作”，实现从&general_tag 到&thread 的地址转换，将它们还原成线程的 PCB 地址后才能使用。这两个线程“标签”定义在新的 thread.c 中，一会儿在介绍时咱们会细说的。

pgdir 是任务自己的页表。线程与进程的最大区别就是进程独享自己的地址空间，即进程有自己的页表，而线程共享所在进程的地址空间，即线程无页表。如果该任务为线程，pgdir 则为 NULL，否则 pgdir 会被赋予页表的虚拟地址，注意此处是虚拟地址，页表加载时还是要被转换成物理地址的，不过这部分内容咱们将来用到时再介绍。

好啦，thread.h 就这点变化，接下来咱们再看 thread.c，请看代码 9-7。

代码 9-7　（project/c9/c/thread/thread.c）

```
…略
10 #define PG_SIZE 4096
11
12 struct task_struct* main_thread;     // 主线程 PCB
13 struct list thread_ready_list;       // 就绪队列
14 struct list thread_all_list;        // 所有任务队列
15 static struct list_elem* thread_tag;// 用于保存队列中的线程结点
16
17 extern void switch_to(struct task_struct* cur, struct task_struct* next);
18
19 /* 获取当前线程 pcb 指针 */
20 struct task_struct* running_thread() {
21     uint32_t esp;
22     asm ("mov %%esp, %0" : "=g" (esp));
23    /* 取 esp 整数部分，即 pcb 起始地址 */
24     return (struct task_struct*)(esp & 0xfffff000);
25 }
26
27 /* 由 kernel_thread 去执行 function(func_arg) */
28 static void kernel_thread(thread_func* function, void* func_arg) {
29 /* 执行 function 前要开中断，
   避免后面的时钟中断被屏蔽，而无法调度其他线程 */
30     intr_enable();
31     function(func_arg);
```

```
32 }
       …略
48 /* 初始化线程基本信息 */
49 void init_thread(struct task_struct* pthread, char* name, int prio) {
50      memset(pthread, 0, sizeof(*pthread));
51      strcpy(pthread->name, name);
52
53      if (pthread == main_thread) {
54 /* 由于把 main 函数也封装成一个线程，
   并且它一直是运行的，故将其直接设为 TASK_RUNNING */
55          pthread->status = TASK_RUNNING;
56      } else {
57          pthread->status = TASK_READY;
58      }
59
60 /* self_kstack 是线程自己在内核态下使用的栈顶地址 */
61      pthread->self_kstack = (uint32_t*)((uint32_t)pthread + PG_SIZE);
62      pthread->priority = prio;
63      pthread->ticks = prio;
64      pthread->elapsed_ticks = 0;
65      pthread->pgdir = NULL;
66      pthread->stack_magic = 0x19870916;    // 自定义的魔数
67 }
68
69 /* 创建一优先级为 prio 的线程，线程名为 name，
   线程所执行的函数是 function(func_arg) */
70 struct task_struct* thread_start(char* name, \
                                    int prio, \
                                    thread_func function, \
                                    void* func_arg) {
71 /* pcb 都位于内核空间，包括用户进程的 pcb 也是在内核空间 */
72      struct task_struct* thread = get_kernel_pages(1);
73
74      init_thread(thread, name, prio);
75      thread_create(thread, function, func_arg);
76
77      /* 确保之前不在队列中 */
78      ASSERT(!elem_find(&thread_ready_list, &thread->general_tag));
79      /* 加入就绪线程队列 */
80      list_append(&thread_ready_list, &thread->general_tag);
81
82      /* 确保之前不在队列中 */
83      ASSERT(!elem_find(&thread_all_list, &thread->all_list_tag));
84      /* 加入全部线程队列 */
85      list_append(&thread_all_list, &thread->all_list_tag);
86
87      return thread;
88 }
89
90 /* 将 kernel 中的 main 函数完善为主线程 */
91 static void make_main_thread(void) {
92 /* 因为 main 线程早已运行，
    * 咱们在 loader.S 中进入内核时的 mov esp,0xc009f000,
93  * 就是为其预留 pcb 的，因此 pcb 地址为 0xc009e000,
    * 不需要通过 get_kernel_page 另分配一页*/
94     main_thread = running_thread();
95     init_thread(main_thread, "main", 31);
96
97 /* main 函数是当前线程，当前线程不在 thread_ready_list 中，
98  * 所以只将其加在 thread_all_list 中 */
99    ASSERT(!elem_find(&thread_all_list, &main_thread->all_list_tag));
100   list_append(&thread_all_list, &main_thread->all_list_tag);
101 }
```

代码 9-7 看上去有点长，但我告诉大家，里面要看的内容并不多，这只是有过改动的部分，全贴出来方便大伙知道在哪里有过变化。

在开头定义了一些全局的数据结构。

第 12 行的“struct task_struct* main_thread”是定义主线程的 PCB，咱们进入内核后一直执行的是 main 函数，其实它就是一个线程，我们在后面会将其完善成线程的结构，因此为其先定义了个 PCB。

调度器要选择某个线程上处理器运行的话，必然要先将所有线程收集到某个地方，这个地方就是线程就绪队列。在第 13 行的“struct list thread_ready_list”便是就绪队列，以后每创建一个线程就将其加到此队列中。

就绪队列中的线程都是可用于直接上处理器运行的，可有时候线程因为某些原因阻塞了，不能放在就绪队列中，但我们得有个地方能找到它，得知道我们共创建了多少线程，为此我们创建了所有（全部）线程队列，它就是第 14 行的 thread_all_list。

在队列中的结点并不是线程的 PCB，而是线程 PCB 中的 tag，即 general_tag 或 all_list_tag，其类型是 struct list_elem。我们对线程的管理都是基于线程 PCB 的，其类型是 struct task_struct，因此必须要将 tag 转换成 PCB。在转换过程中需要记录 tag 的值，因此在第 15 行定义了全局变量 thread_tag 来存储。（悄悄说一句，其实可以用局部变量来完成，形式不限。）

接下来用 extern 声明了外部函数 switch_to，在后面会介绍。

新增了个函数 running_thread，它的功能是返回线程的 PCB 地址。原理很简单，各个线程所用的 0 级栈都是在自己的 PCB 当中，因此取当前栈指针的高 20 位作为当前运行线程的 PCB。里面都是简单的内联汇编，相信大伙儿都能看懂。

本版本的 kernel_thread 函数有了很重要的变化，在函数体中增加了开中断的函数 intr_enable，它是任务调度的保证。原因是我们的任务调度机制基于时钟中断，由时钟中断这种“不可抗力”来中断所有任务的执行，借此将控制权交到内核手中，由内核的任务调度器 schedule（后面有小节专门论述 schedule）考虑将处理器使用权发放到某个任务的手中，下次中断再发生时，权利将再被回收，周而复始，这样便保证操作系统不会被“架空”，而且保证所有任务都有运行的机会。

线程的首次运行是由时钟中断处理函数调用任务调度器 schedule 完成的，进入中断后处理器会自动关中断，因此在执行 function 前要打开中断，否则 kernel_thread 中的 function 在关中断的情况下运行，也就是时钟中断被屏蔽了，再也不会调度到新的线程，function 会独享处理器。

在介绍 init_thread 之前，有些事情要和大伙交待清楚。其实我们从开机到创建第一个线程前，程序都有个执行流，这个执行流带我们从 BIOS 到 mbr 到 loader 到 kernel，其实它就是我们所说的主线程。为此我们还在 loader 中把 esp 置为 0xc009f000，这是有意为之的设计，意图是把 0xc009e000 作为主线程的 PCB。现在我们要创建新线程了，并且打开了时钟中断，时钟中断的处理函数会判断当前线程所用的栈是否会破坏了线程信息，也就是判断 thread->stack_magic 是否等于 0x19870916。可是新创建的线程在首次运行前一直是主线程在跑，但此时主线程还没有身份证，也就是还没有 PCB，因此 main_thread->stack_magic 就是错误的值，所以我们提前调用 make_main_thread 函数为主线程赋予了 PCB，到后面都有介绍。

好啦，现在大伙儿知道主线程 main_thread 已经有 PCB 了，它现在是有“身份证”的线程了，所以 init_thread 函数也有了一些改变，第 53～58 行，我们加入了对主线程 main_thread 的判断，如果待初始化的线程是主线程，也就是代码“if (pthread == main_thread)”的判断，那就将线程的状态置为 TASK_RUNNING，因为主线程早已经在运行了。否则的话，就表示待初始化的线程不是主线程，它们的状态为准备运行，因此置其状态为 TASK_READY。

在第 64 行的“pthread->elapsed_ticks = 0”，表示线程尚未执行过。线程没有自己的地址空间，因此第 65 行将线程的页表置空“pthread->pgdir = NULL”。

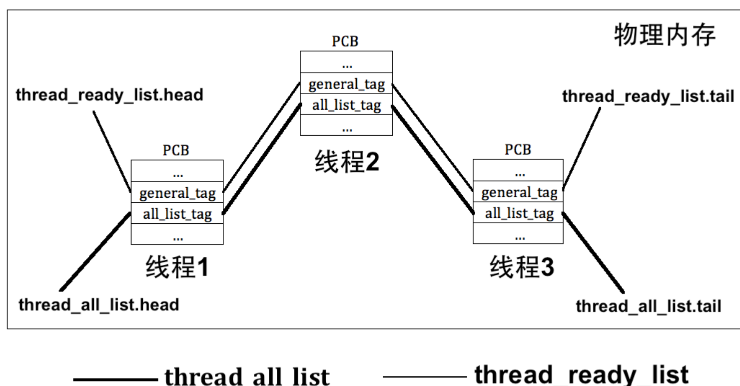
thread_start 函数也有了变化，此函数是创建线程的入口，主要变化是在执行完 init_thread 和 thread_create 后，通过 list_append 函数把新创建的线程加入了就绪队列和全部线程队列。拿就绪队列来说，在加入队列之前，按理说线程不应该出现在就绪队列 thread_ready_list 中，因此在加入队列之前，先通过 ASSERT 来保证这一点。

ASSERT 中调用的是 elem_find(&thread_ready_list, &thread->general_tag)，第 1 个参数是就绪队列的地址&thread_ready_list，第 2 个参数是新线程的标签 general_tag 的地址。elem_find 的功能是在队列中找结点，如果找到就返回 1，否则返回 0。由此可见，队列中的结点就是线程 PCB 中的成员 general_tag。

这一点可以在接下来的 list_append 中得到验证，list_append 的功能就是在队列 thread_ready_list 中加入新创建线程的 general_tag 成员，因此传入的参数是 general_tag 的地址，即“&thread->general_tag”。一定要清楚，链表（队列）中的结点并不是 PCB，因此这里并不是把 PCB 加到队列中，我们链表的结点类

型是 struct list_elem，只能将 struct list_elem 类型的结点插入到队列中，而 PCB 中 general_tag 的类型就是 struct list_elem，这是在设计 PCB 时有意安排的。这样做是有好处的，struct list_elem 类型中只有两个指针成员：struct list_elem* prev 和 struct list_elem* next，因此它作为结点的话，结点尺寸就 8 字节，整个队列显得轻量小巧，如果换成 PCB 做结点，尺寸也太大了。

线程在内存中的位置是散落的，由不同的链表将它们各自的 general_tag 和 all_list_tag 串联起来，从而形成队列。线程在队列中的组织结构如图 9-11 所示。



▲图 9-11　线程在队列中的组织结构

接下来的函数是前面提到过的 make_main_thread，它在主线程的 PCB 中写入线程信息。正如前面介绍，主线程已经跑起来了，所以不需要再为其申请页安装其 PCB，也不需要再通过 thread_create 构造它的线程栈，只需要通过 init_thread 填充其名称和优先级。咱们只有两个队列，“就绪队列”只存储准备运行的线程，“全部队列”存储所有线程，包括就绪的、阻塞的、正在执行的，因此，只需要将主线程加入到全部队列 thread_all_list 中。

本节到此结束，下节咱们介绍和调度相关的代码。

9.4.2　任务调度器和任务切换

本节要实现调度器和任务切换，调度器的工作就是根据任务的状态将其从处理器上换上换下，任务的状态是咱们定义的，因此定义任务状态目的就是为了方便咱们设计任务调度的方法。您看，操作系统设计并非死板，每个数据的定义都是为了“自圆其说”。

调度器主要任务就是读写就绪队列，增删里面的结点，结点是线程 PCB 中的 general_tag，“相当于”线程的 PCB，从队列中将其取出时一定要还原成 PCB 才行。

咱们的调度原理比较简单，看看我们是如何自圆其说的。

线程每次在处理器上的执行时间是由其 ticks 决定的，我们在初始化线程的时候，已经将线程 PCB 中的 ticks 赋值为 prio，优先级越高，ticks 越大。每发生一次时钟中断，时钟中断的处理程序便将当前运行线程的 ticks 减 1。当 ticks 为 0 时，时钟的中断处理程序调用调度器 schedule，也就是该把当前线程换下处理器了，让调度器选择另一个线程上处理器。

调度器是从就绪队列 thread_ready_list 中“取出”上处理器运行的线程，所有待执行的线程都在 thread_ready_list 中，我们的调度机制很简单，就是 Round-Robin Scheduling，俗称 RR，即轮询调度，说白了就是让候选线程按顺序一个一个地执行，咱们就是按先进先出的顺序始终调度队头的线程。注意，这里说的是“取出”，也就是从队列中弹出，意思是说队头的线程被选中后，其结点不会再从就绪队列 thread_ready_list 中保存，因此，按照先入先出的顺序，位于队头的线程永远是下一个上处理器运行的线程。就绪队列 thread_ready_list 中的线程都属于运行条件已具备，但还在等待被调度运行的线程，因此 thread_ready_list 中的线程的状态都是 TASK_READY。而当前运行线程的状态为 TASK_RUNNING，它仅

保存在全部队列 thread_all_list 当中。

调度器 schedule 并不仅由时钟中断处理程序来调用，它还有被其他函数调用的情况，比如后面要介绍的函数 thread_block。因此，在 schedule 中要判断当前线程是出于什么原因才“沦落到”要被换下处理器的地步。是线程的时间片到期了？还是线程时间片未到，但它被阻塞了，以至于不得不换下处理器？其实这就是查看线程的状态，如果线程的状态为 TASK_RUNNING，这说明时间片到期了，将其 ticks 重新赋值为它的优先级 prio，将其状态由 TASK_RUNNING 置为 TASK_READY，并将其加入到就绪队列的末尾。如果状态为其他，这不需要任何操作，因为调度器是从就绪队列中取出下一个线程，而当前运行的线程并不在就绪队列中。

调度器按照队列先进先出的顺序，把就绪队列中的第 1 个结点作为下一个要运行的新线程，将该线程的状态置为 TASK_RUNNING，之后通过函数 switch_to 将新线程的寄存器环境恢复，这样新线程便开始执行。

因此，完整的调度过程需要三部分的配合。

（1）时钟中断处理函数。

（2）调度器 schedule。

（3）任务切换函数 switch_to。

以上只是个理论框架，咱们按照这个顺序在后面几节介绍具体的内容。

1．注册时钟中断处理函数

大伙儿都知道，中断处理函数一定得在中断描述符表中，与此中断向量对应的中断描述符里提前注册好才能用。咱们自己的中断处理逻辑是由 kernel.S 提供统一的中断入口，即中断向量 0～30 全是用统一的中断处理程序“模板”，在该模板中通过中断向量号调用中断处理程序数组 idt_table 中的 C 版本的处理程序，也就是文件 kernel.S 中代码 call [idt_table + %1*4]的作用。因此，为设备注册中断处理程序的工作变得很简单，我们不用去修改中断描述符，直接把中断向量作为数组下标，去修改 idt_table[中断向量]数组元素即可。

大伙儿还记得，之前的时钟中断处理函数还是用通用的函数来处理的，即 general_intr_handler，此函数作为默认的中断处理函数，即某个中断源没有中断处理程序时才用它来代替。不过为了调度方便，general_intr_handler 也改进了一小下，那咱们先来段小插曲，看看代码 9-8。

代码 9-8　（project/c9/c/kernel/interrupt.c）

```
 74 /* 通用的中断处理函数，一般用在异常出现时的处理 */
 75 static void general_intr_handler(uint8_t vec_nr) {
 76     if (vec_nr == 0x27 || vec_nr == 0x2f) {
            // 0x2f 是从片 8259A 上的最后一个 irq 引脚，保留
 77         return; //IRQ7 和 IRQ15 会产生伪中断（spurious interrupt），无需处理
 78     }
 79 /* 将光标置为 0，从屏幕左上角清出一片打印异常信息的区域，方便阅读 */
 80     set_cursor(0);
 81     int cursor_pos = 0;
 82     while(cursor_pos < 320) {
 83         put_char(' ');
 84         cursor_pos++;
 85     }
 86
 87     set_cursor(0);         // 重置光标为屏幕左上角
 88     put_str("!!!!!!!     excetion message begin  !!!!!!!!\n");
 89     set_cursor(88);        // 从第 2 行第 8 个字符开始打印
 90     put_str(intr_name[vec_nr]);
 91     if (vec_nr == 14) {    // 若为 Pagefault，将缺失的地址打印出来并悬停
 92          int page_fault_vaddr = 0;
 93          asm ("movl %%cr2, %0" : "=r" (page_fault_vaddr));
                               // cr2 是存放造成 page_fault 的地址

 94          put_str("\npage fault addr is ");put_int(page_fault_vaddr);
 95     }
 96     put_str("\n!!!!!!!     excetion message end    !!!!!!!!\n");
 97    // 能进入中断处理程序就表示已经处在关中断情况下
 98    // 不会出现调度进程的情况。故下面的死循环不会再被中断
 99     while(1);
100 }
```

较之前的版本相比，此版本内容虽然还是很少，但改进的地方可不少，这里有个新的函数 set_cursor，它接受一个参数，就是光标值（光标值范围是 0～1999）。它的功能就是设置光标的值，其函数实现就是文件 print.S 中函数 put_char 的.set_cursor 部分，因此这里不再单独贴出。

为什么需要这个函数呢？有时候屏幕上的内容太多了，打印的提示信息不利用阅读。甚至有时候的异常是由光标错误值引发的，因此必须在异常中将光标位置纠正，否则在通过 put_str 输出报错信息的时候，错误的光标值将再次导致异常，可能会造成异常的死循环，因此更谈不上输出异常信息了，所以稳妥起见，咱们还是在异常处理程序中将其置为正确的值。

程序运行时最后输出的有用信息一般都在屏幕最下方，咱们最好不占用下方的屏幕，而是将异常信息输出在屏幕左上角，因此先调用“set_cursor(0)”将光标置为 0。

为方便阅读，在输出异常信息之前先通过 while 循环清空 4 行内容，也就是填入了 4 行空格，一行字符数是 80 个，因此共 320 个空格输出的循环。接着再次调用“set_cursor(0)”将光标置为 0，也就是屏幕左上角，这样异常信息将在刚刚清空的地方输出。

此外还加进了 Pagefault 的处理。Pagefault 就是通常所说的缺页异常，它表示虚拟地址对应的物理地址不存在，也就是虚拟地址尚未在页表中分配物理页，这样会导致 Pagefault 异常。导致 Pagefault 的虚拟地址会被存放到控制寄存器 CR2 中，我们加入的内联汇编代码就是让 Pagefault 发生时，将寄存器 cr2 中的值转储到整型变量 page_fault_vaddr 中，并通过 put_str 函数打印出来。因此，如果程序运行过程中出现异常 Pagefault 时，将会打印出导致 Pagefault 出现的虚拟地址。

以后各设备都会注册自己的中断处理程序，不会再使用 general_intr_handler。我们没有为各种异常注册相应的中断处理程序，这里是用 general_intr_handler 作为通用的中断处理程序来“假装处理”异常的，general_intr_handler 的功能是打印调试信息并将程序停止，也就是提醒咱们出问题了，该调试了，因此只要执行到 general_intr_handler 中就表示出了某些异常。

处理器进入中断后会自动把标志寄存器 eflags 中的 IF 位置 0，即中断处理程序在关中断的情况下运行。因此，通过后面的 while(1)语句便能够将程序悬停在此，这样便于观察报错信息。

general_intr_handler 就介绍完了，咱们回到正题。既然时钟中断有专门的用途了，我们就要为其注册专门的处理函数。我们先看看它的内容是什么，一会儿再来说注册的事。

先看代码 9-9，这是改进后的时钟中断处理程序。

代码 9-9　（project/c9/c/device/timer.c）

```
…略
17 uint32_t ticks;         // ticks 是内核自中断开启以来总共的嘀嗒数
…略
33 /* 时钟的中断处理函数 */
34 static void intr_timer_handler(void) {
35     struct task_struct* cur_thread = running_thread();
36
37     ASSERT(cur_thread->stack_magic == 0x19870916); // 检查栈是否溢出
38
39     cur_thread->elapsed_ticks++;   // 记录此线程占用的 cpu 时间
40     ticks++;    //从内核第一次处理时间中断后开始至今的滴哒数，内核态和用户态总共的嘀哒数
41
42     if (cur_thread->ticks == 0) {   // 若进程时间片用完，就开始调度新的进程上 cpu
43         schedule();
44     } else {                   // 将当前进程的时间片-1
45         cur_thread->ticks--;
46     }
47 }
48
49 /* 初始化 PIT8253 */
50 void timer_init() {
51     put_str("timer_init start\n");
52     /* 设置 8253 的定时周期，也就是发中断的周期 */
53     frequency_set(CONTRER0_PORT, \
   COUNTER0_NO, \
   READ_WRITE_LATCH, \
   COUNTER_MODE, \
```

```
   COUNTER0_VALUE);
54     register_handler(0x20, intr_timer_handler);
55     put_str("timer_init done\n");
56 }
```

计算机中是用时钟来表示节奏，咱们也有“嘀嗒”来表示中断次数，借以表示时间。

代码 9-9 的开头定义了 ticks，它用来保存系统自开中断以来所运行的嘀嗒数，类似于系统运行时长的概念，以后在写用户程序的时候也许会用到。

新创建的时钟处理函数是 intr_timer_handler，它先通过“running_thread()”获取当前正在运行的线程，将其赋值给 PCB 指针 cur_thread。

第 37 行通过 ASSERT 来判断 stack_magic 是否等于 0x19870916，也就是检查栈是否溢出，破坏了线程信息。正常情况下不会出现栈溢出，因此只要 ASSERT 报警就表示哪里出了问题，不用处理了，直接调试吧。

第 39 行“cur_thread->elapsed_ticks++”，将线程总执行的时间加 1。

第 40 行的 ticks++将系统运行时间加 1，实际上这就是中断发生的次数。

每个线程在处理器上运行期间都会有很多次时钟中断发生，每次中断处理程序都会将线程的时间片 ticks 减 1。

第 42～46 行判断当前线程的时间片 ticks 是否用完了，如果 ticks 等于 0，说明当前线程 cur_thread 时间片耗尽，该下处理器了，此时调用 schedule 函数。否则将当前线程的时间片 ticks 减 1。

之后退出中断处理程序，也就是退出中断，让当前线程 cur_thread 继续执行。

在下面的 timer_init 中，我们加入了注册时钟中断处理程序的代码，即“register_handler(0x20, intr_timer_handler)”，timer_init 是由 init_all 调用的，它在内核运行开始处执行的，故，时钟中断会被提前注册好。

下面看看 register_handler 的实现原理，见代码 9-10。

代码 9-10　（project/c9/c/kernel/interrupt.c）

```
…略
163 /* 在中断处理程序数组第 vector_no 个元素中
    注册安装中断处理程序 function */
164 void register_handler(uint8_t vector_no, intr_handler function) {
165 /* idt_table 数组中的函数是在进入中断后根据中断向量号调用的
166   * 见 kernel/kernel.S 的 call [idt_table + %1*4] */
167    idt_table[vector_no] = function;
168 }
…略
```

register_handler 接受两个参数，vector_no 是中断向量号，function 是中断处理程序。功能是在中断处理程序数组第 vector_no 个元素中注册安装中断处理程序 function。

再一看函数实现，就是“idt_table[vector_no] = function”，太简单了是吧？哈哈，好，本节到此结束。

2．实现调度器 schedule

本节实现任务切换的第二个重要环节，调度器。

虽然本节为单独的小节，但还是属于 thread.c 中的代码，其实代码 9-11 是代码 9-7 的延续。

代码 9-11　（project/c9/c/thread/thread.c）

```
103 /* 实现任务调度 */
104 void schedule() {
105
106    ASSERT(intr_get_status() == INTR_OFF);
107
108    struct task_struct* cur = running_thread();
109    if (cur->status == TASK_RUNNING) {
           // 若此线程只是 cpu 时间片到了，将其加入到就绪队列尾
110         ASSERT(!elem_find(&thread_ready_list, &cur->general_tag));
111         list_append(&thread_ready_list, &cur->general_tag);
112         cur->ticks = cur->priority;
           // 重新将当前线程的 ticks 再重置为其 priority
113        cur->status = TASK_READY;
```

```
114     } else {
115        /* 若此线程需要某事件发生后才能继续上 cpu 运行，
116        不需要将其加入队列，因为当前线程不在就绪队列中 */
117     }
118
119     ASSERT(!list_empty(&thread_ready_list));
120     thread_tag = NULL;    // thread_tag 清空
121 /* 将 thread_ready_list 队列中的第一个就绪线程弹出，
    准备将其调度上 cpu */
122     thread_tag = list_pop(&thread_ready_list);
123     struct task_struct* next = elem2entry(struct task_struct, \
        general_tag, \ thread_tag);
124     next->status = TASK_RUNNING;
125     switch_to(cur, next);
126 }
127
```

schedule 的原理之前已介绍，它的功能是将当前线程换下处理器，并在就绪队列中找出下个可运行的程序，将其换上处理器。schedule 主要内容就是读写就绪队列，因此它不需要参数。

第 108 行，通过 running_thread()获取了当前运行线程的 PCB，将其存入 PCB 指针 cur 中（它就是代表 cur_thread，只是简写了），接下来的判断都是基于 cur 的，可见 PCB 对任务来说确实是非常重要的“身份证”，操作系统通过此身份证来了解进程的一切。

接下来分两种情况来考虑，如果当前线程 cur 的时间片到期了，就将其通过 list_append 函数重新加入到就绪队列 thread_ready_list。由于此时它的时间片 ticks 已经为 0，为了下次运行时不至于马上被换下处理器，将 ticks 的值再次赋值为它的优先级 prio，使其下次能够“满血复活”，最后将 cur 的状态 status 置为 TASK_READY。

如果当前线程 cur 并不是因为时间片到期而被换下处理器，肯定是由于某种原因被阻塞了（比如对 0 值的信号量进行 P 操作就会让线程阻塞，到同步机制时会介绍），这时候不需要处理就绪队列，因为当前运行线程并不在就绪队列中，咱们下面来看当前运行的线程是如何从就绪队列中“出队”的。

我们尚未实现 idle 线程，因此有可能就绪队列为空，为避免这种无线程可调度的情况，暂时用“ASSERT (!list_empty(&thread_ready_list))”来保障。

下面是将 thread_tag 置为 NULL，由于它是全局变量，个人习惯使用前先将其清空，避免在下面赋值失败时排查起来困难。

接下来通过“thread_tag = list_pop(&thread_ready_list)”从就绪队列中弹出一个可用线程并存入 thread_tag。注意啦，thread_tag 并不是线程，它仅仅是线程 PCB 中的 general_tag 或 all_list_tag，要获得线程的信息，必须将其转换成 PCB 才行，因此我们用到了宏 elem2entry，是时候好好说说它了。

elem2entry 定义在 list.h 中，我把和它相关的宏都贴过来了。

```
5 #define offset(struct_type,member) (int)(&((struct_type*)0)->member)
6 #define elem2entry(struct_type, struct_member_name, elem_ptr) \
7  (struct_type*)((int)elem_ptr - offset(struct_type, struct_member_name))
```

宏 elem2entry 接受三个参数，咱们从后往前说。

参数 elem_ptr 是待转换的地址，它属于某个结构体中某个成员的地址，参数 struct_member_name 是 elem_ptr 所在结构体中对应地址的成员名字，也就是说参数 struct_member_name 是个字符串，参数 struct_type 是 elem_ptr 所属的结构体的类型。宏 elem2entry 的作用是将指针 elem_ptr 转换成 struct_type 类型的指针，其原理是用 elem_ptr 的地址减去 elem_ptr 在结构体 struct_type 中的偏移量，此地址差便是结构体 struct_type 的起始地址，最后再将此地址差转换为 struct_type 指针类型。这里涉及到了另外一个宏 offset，一会儿咱们再讨论它。

一般的转型转换，只是改变了数据类型，并不改变地址，不同的数据类型仅仅是告诉编译器在同一地址处连续获取多少字节的数据。比如：

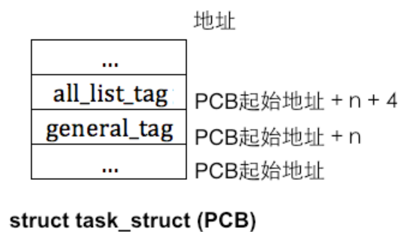
```
int four_bytes = 0x12345678;
char one_bytes = (char)four_bytes;
```

Intel 是小端字节序，因此单字节变量 one_bytes 的值为 0x78。

我们这里要做的转换，不仅包括类型，还涉及到地址的转换。从队列中弹出的结点元素并不能直接用，因为咱们链表中的结点并不是 PCB，而是 PCB 中的 general_tag 或 all_list_tag，需要将它们转换成所在的 PCB 的地址。比如咱们的办公地点是某大厦七层，有人要给咱们寄快递，咱们不能再告诉人家楼层了，必须要告诉人家大厦的地址。这种由楼层地址到大厦地址的转换就类似由以上两个 tag 的地址到 PCB 的地址转换。

所以，整个转换过程要分为两步，先完成地址转换，再完成类型转换。另外，为叙述方便，暂时将 general_tag 和 all_list_tag 统称为 x_tag。

咱们看看 x_tag 在 PCB 中的布局，如图 9-12 所示。



▲图 9-12　x_tag 在 PCB 中的分布

下面开始讨论如何完成地址转换。

无论 PCB 的地址是多少，x_tag 在 PCB 中的偏移量是固定的，如果要将它们的地址转换成 PCB 地址的话，有两种方法。

（1）PCB 是在自然页的起始地址，也就是 PCB 的地址=0xfffff000&(&(PCB.general_tag))。

（2）用 x_tag 的地址减去它们在 PCB 中的偏移量。

我们在函数 running_thread 中利用栈指针 esp 获取线程 PCB 就是用的第 1 种方法。似乎这种方法最简单，但您不想试试第 2 种方法吗？看上去似乎有点繁琐，但也更有魅力，多学习一种方法不是更好吗？这让我想起了那句话：折腾是对生命的尊重，咱们就用第 2 种方法吧。

访问结构体成员的两种方法是“结构体变量.成员”和“结构体指针变量->成员”。它们的访问原理是“结构体变量的地址+成员的偏移量”，这种寻址方式相当于“基址+变址”，其中“结构体变量的地址”相当于基址，“成员的偏移量”相当于变址，可以近似认为访问结构体成员的方法等效于“基址->变址”，结构体成员的地址等于“&（基址->变址）”。

还是看图 9-12，&PCB 相当于基址。general_tag 在 PCB 中的偏移量 = &(PCB.general_tag)–&PCB = n，这里的&PCB 恰恰是咱们最终要求解的，因此这是绕了个圈子又回来了。但咱们此时关注的是偏移量 n，倘若令基址&PCB 的值等于 0，&(PCB.general_tag)不就等于偏移量 n 了吗。

因此，我们有办法直接获得 x_tag 的偏移量，看看宏 offset 的定义您就清楚了，它接受两个参数，struct_type 是结构体类型，member 是结构体成员的名字，其核心代码“&((struct_type*)0)->member”则为结构体成员 member 在结构体中的偏移量。

有了地址后就好说了，咱们还差类型转换没做，现在再回头看看宏 elem2entry 的原理，它将转换分为两步。

（1）用结构体成员的地址减去成员在结构体中的偏移量，先获取到结构体起始地址。

（2）再通过强制类型转换将第 1 步中的地址转换成结构体类型。

好啦，宏 elem2entry 就介绍完了，咱们继续说代码，插了这么大一段插曲有点不适应了。

在代码 9-11 的第 123 行通过 elem2entry 获得了新线程的 PCB 地址，将其赋值给 next，紧接着通过“next-> status = TASK_RUNNING”将新线程的状态置为 TASK_RUNNING，这表示新线程 next 可以上处理器了，于是准备切换寄存器映像，这是通过调用 switch_to 函数完成的，调用形式为“switch_to(cur, next)”，意为将线程 cur 的上下文保护好，再将线程 next 的上下文装载到处理器，从而完成了任务切换。有关这个函数咱们下节再说。

3．实现任务切换函数 switch_to

本节是任务切换的最后一个环节，任务切换，这是由汇编函数 switch_to 完成的工作。

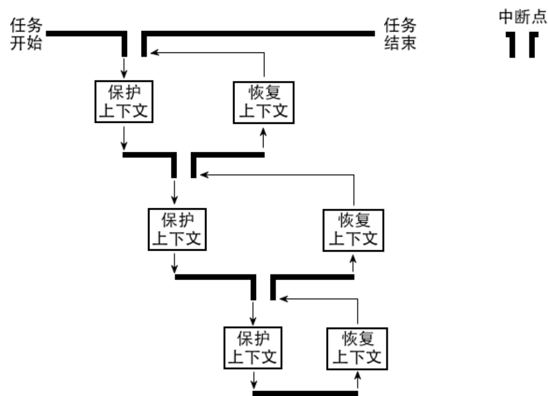
操作系统存在的主要目的就是方便人们编写程序，为了方便、安全等原因，操作系统把和硬件或安全相关的工作自己独自揽下来，当然这并不是出于操作系统的“奉献”精神，而是把这么重要的工作交给别人它不放心，操作系统只相信自己。这样一来，程序所做的完整工作可以分为两部分，一部分是“重要工作”，这由操作系统代码来完成，另一部分是“普通工作”，这由用户代码完成。比如某程序想在硬盘上创建文件，用户代码的工作只是提供文件名及相关属性，这属于普通工作，将文件在硬盘上创建属于重要工作，这是由操作系统完成的。实际上，完整的程序就也因此分为两部分，一部分是做重要工作的内核级代

码，另一部分就是做普通工作的用户级代码。所以，“完整的程序=用户代码+内核代码”。而这个完整的程序就是我们所说的任务，也就是线程或进程，也就是说，任务在执行过程中会执行用户代码和内核代码，当处理器处于低特权级下执行用户代码时我们称之为用户态，当处理器进入高特权级执行到内核代码时，我们称之为内核态，当处理器从用户代码所在的低特权级过渡到内核代码所在的高特权级时，这称为陷入内核。因此一定要清楚，无论是执行用户代码，还是执行内核代码，这些代码都属于这个完整的程序，即属于当前任务，并不是说当前任务由用户态进入内核态后当前任务就切换成内核了，这样理解是不对的。任务与任务的区别在于执行流一整套的上下文资源，这包括寄存器映像、地址空间、IO 位图等，在将来介绍任务状态段 TSS 之后，您就会了解这套上下文资源恰恰就是 TSS 结构中的内容，拥有这些资源才称得上是任务。因此，处理器只有被新的上下文资源重新装载后，当前任务才被替换为新的任务，这才叫任务切换。当任务进入内核态时，其上下文资源并未完全替换，只是执行了“更厉害”的代码。这有点像咱们在游戏中打怪，用不同的武器打不同的怪，游戏的人物角色始终没有变。

用户代码完成的普通工作当然就和内核所负责的“重要工作”没什么关系，所以即使出错了也不会对计算机造成致命的伤害，当程序需要做“重要工作”时，就通过系统调用让内核帮忙完成，也就是执行“重要工作”的内核代码。因此，任何一个程序要想完整地做一件事，它必须执行两部分代码，这下您应该清楚了，我们平时所写的用户程序如果需要内核帮忙，它顶多只是个半成品（一点系统调用都不需要的用户程序才算完整的成品），它在执行过程中需要与内核代码组合成完整的程序。有关这一点大伙儿可以参考图 5-18——进程共享操作系统和图 5-19——完整的程序。

好啦，热身结束，步入正题。为什么要保护任务的上下文？

每个任务都有个执行流，这都是事先规划好的执行路径，按道理应该是从头执行到结束。不过实际的情况是执行流经常被临时改道，突然就执行了规划外的指令，这在多任务系统中是很正常的，因为操作系统是由中断驱动的，每一次中断都将使处理器放下手头的工作转去执行中断处理程序。为了在中断处理完成后能够恢复任务原有的执行路径，必须在执行流被改变前，将任务的上下文保护好。执行流被改变后，在其后续的执行过程中还可能会再次发生被改变“流向”的情况，也就是说随着执行的深入，这种改变的深度很可能是多层的。如果希望将来能够返回到本层的执行流，依然要在改变前保护好本层的上下文。总之，凡是涉及到执行流的改变，不管被改变了几层，为了将来能够恢复到本层继续执行，必须在改变发生前将本层执行流的上下文保护好。因此，执行流被改变了几层就要做几次上下文保护，如图 9-13 所示。



▲图 9-13 上下文逐层保护

在咱们的系统中，任务调度是由时钟中断发起，由中断处理程序调用 switch_to 函数实现的。假设当前任务在中断发生前所处的执行流属于第一层，受时钟中断的影响，处理器会进入中断处理程序，这使当前的任务执行流被第一次改变，因此在进入中断时，我们要保护好第一层的上下文，即中断前的任务状态。之后在内核中执行中断处理程序，这属于第二层执行流。当中断处理程序调用任务切换函数 switch_to 时，当前的中断处理程序又要被中断，因此要保护好第二层的上下文，即中断处理过程中的任务状态。

因此，咱们系统中的任务调度，过程中需要保护好任务两层执行流的上下文，这分两部分来完成。

第一部分是进入中断时的保护，这保存的是任务的全部寄存器映像，也就是进入中断前任务所属第一层的状态，这些寄存器映像相当于任务中用户代码的上下文。

这些寄存器是由 kernel.S 中定义的中断处理入口程序 intr%1entry 来保护的，里面是一些 push 寄存器的指令，这是由汇编下的宏“%macro VECTOR 2”定义的，因此前面在介绍注册中断处理程序时曾称之为中断处理程序“模板”。当把这些寄存器映像恢复到处理器中后，任务便完全退出中断，继续执行自己的代码部分。换句话说，当恢复寄存器后，如果此任务是用户进程，任务就完全恢复为用户程序继续在用户态下

执行，如果此任务是内核线程，任务就完全恢复为另一段被中断执行的内核代码，依然是在内核态下运行。

第二部分是保护内核环境上下文，根据 ABI，除 esp 外，只保护 esi、edi、ebx 和 ebp 这 4 个寄存器就够了。这 4 个寄存器映像相当于任务中的内核代码的上下文，也就是第二层执行流，此部分只负责恢复第二层的执行流，即恢复为在内核的中断处理程序中继续执行的状态。下面需要结合咱们的实现来解释为什么这么做了。

当任务开始执行内核代码后，任务在内核代码中的执行路径由这 4 个寄存器决定，将来恢复这 4 个寄存器，也只是让处理器继续执行任务中的内核代码，并不是让任务恢复到中断前，依然还是在内核中。那为什么还要保护这几个寄存器呢？因为这几个寄存器的值会让处理器把程序执行到内核代码的结束处，在那里可以用第一部分中保护的全部寄存器映像来恢复任务，从而退出中断，使任务彻底恢复为进入中断前的状态。另外，其实这 4 个寄存器主要是用来恢复主调函数的环境，只是当前我们在讨论内核函数。

中断发生时，当前运行的任务（线程或用户进程）被打断，随后会去执行中断处理程序，不管当前任务在中断前的特权级是什么，执行中断处理程序时肯定都是 0 特权级。现在咱们已经达成共识，任务的代码包括用户代码+内核代码，即使是 3 特权级的用户进程进入中断后，当前的任务依然是进入中断前的那个任务，只是此任务目前陷入了内核态，执行的是内核的代码而已，并不是我们平时自己写的那部分用户代码，因此进入中断后所执行的一切内核代码也依然属于当前任务，只是由内核来提供这一部分而已。

内核代码也是人为设计出来的，这套代码保证了任务进入中断后也能顺利退出中断并恢复到中断前的状态。强调了这么久，无非是想和大家说，内核的代码和您自己写的代码都属于同一个任务，都拥有这同一套上下文资源，无论任务是出于何种原因身处内核，作为一个能够“自圆其说”的系统，既然能够让任务进到内核，就同样得能让任务从内核中走出去，因此任务在进入内核后，无论中间过程有多少分支，最终一定会走到出口，从而退出内核中断。

虽然内核代码是任务的一部分，但任务不可能老留在内核中做客，毕竟这种中断只是临时的，它还得赶紧回去忙其他事呢，因此内核代码有责任包含使任务回去的“出口”。咱们是在内核中实现线程机制的，因此任务切换由内核代码完成，这表示当前任务还未执行到“出口”就会被换下处理器停止执行，“出口”在剩下未执行的代码中，待将来再次被调度到处理器上才会继续执行，才会找到“出口”。因此，为了任务在将来再次被换上处理器时能够顺利地找到退出中断的出口，我们更有理由在任务切换前把任务的内核上下文保护好，这样当该任务再次被换上处理器时，才能继续把剩下的内核代码执行完，任务才能找到回去的路，也就是返回到进入中断前的状态，继续执行未完成第一层执行流的工作。我们所提的“出口”，是指 kernel.S 中的 intr_exit，这是退出中断的出口，中断处理完成后，执行流程会通过 jmp intr_exit 跳转到此，此处的指令会用进入中断时保护的寄存器映像装载处理器，从而彻底走出中断，恢复任务。

总结：

（1）上下文保护的第一部分负责保存任务进入中断前的全部寄存器，目的是能让任务恢复到中断前。

（2）上下文保护的第二部分负责保存这 4 个寄存器：esi、edi、ebx 和 ebp，目的是让任务恢复执行在任务切换发生时剩下尚未执行的内核代码，保证顺利走到退出中断的出口，利用第一部分保护的寄存器环境彻底恢复任务。

好啦，扯得够多了，任务上下文保护的第一部分已经在 kernel.S 中由 intr%1entry 完成，咱们现在要完成第二部分。给大伙儿上代码，本节在 thread 目录下创建了 switch.S 文件，其中的函数 switch_to 用于任务切换，详情请看代码 9-12。

代码 9-12 （project/c9/c/ thread/switch.S）

```
 1 [bits 32]
 2 section .text
 3 global switch_to
 4 switch_to:
 5     ;栈中此处是返回地址
 6     push esi
 7     push edi
 8     push ebx
 9     push ebp
10
```

```
11     mov eax, [esp + 20]; 得到栈中的参数cur，cur = [esp+20]
12     mov [eax], esp      ; 保存栈顶指针esp. task_struct的self_kstack字段
13                          ; self_kstack在task_struct中的偏移为0
14                         ; 所以直接往thread开头处存4字节便可

15     ;-------  以上是备份当前线程的环境，下面是恢复下一个线程的环境  ---------

16     mov eax, [esp + 24]    ; 得到栈中的参数next，next = [esp+24]
17     mov esp, [eax]    ; pcb的第一个成员是self_kstack成员
       ; 它用来记录0级栈顶指针，被换上cpu时用来恢复0级栈
18     ; 0级栈中保存了进程或线程所有信息，包括3级栈指针
19     pop ebp
20     pop ebx
21     pop edi
22     pop esi
23     ret    ; 返回到上面switch_to下面的那句注释的返回地址，
24             ; 未由中断进入，第一次执行时会返回到kernel_thread
```

switch.S 其实就写了一个函数 switch_to，此函数接受两个参数，第 1 个参数是当前线程 cur，第 2 个参数是下一个上处理器的线程，此函数的功能是保存 cur 线程的寄存器映像，将下一个线程 next 的寄存器映像装载到处理器。

程序第 3 行 global switch_to 将函数 switch_to 导出为全局符号，这样 thread.c 中的 schedule 便能够使用它了。

第 6～9 行是遵循 ABI 原则，保护好 esi、edi、ebx、ebp 寄存器，在函数的开头进行寄存器保护是个好习惯，避免后面的操作会影响它们的值。栈是自高地址向低地址扩展的，因此这 4 个 push 操作步骤与线程栈 struct thread_stack 的结构是逆序的。顺便强调一句，线程栈 struct thread_stack 仅仅是个数据结构，是个存储数据的格式，表述了结构中各成员的存储顺序，我们仅仅按照此格式中数据成员的顺序来压栈，所以它并不一定在某个固定的内存位置。

我们先看下此时栈中的情况，如图 9-14 所示。

任务切换时的栈

地址(高)	
...	
next	+6*4
cur	+5*4
返回地址	+4*4
esi	+3*4
edi	+2*4
ebx	+1*4
ebp	←ESP
...	

▲图 9-14　switch_to 中的栈

图 9-14 中最下面的 4 个寄存器是咱们进入 switch_to 时压入的，咱们要关注的是 switch_to 的两个参数，cur 和 next 的位置。

在 PCB 结构 struct task_struct 中，第一个成员是 self_kstack，它用来记录每个线程自己的栈顶指针。我们已经知道，任务在内核中的寄存器映像是保存在栈中的，这正是进入 switch_to 函数时立即把那 4 个寄存器入栈的原因。任务在下次被调度运行时，还得把寄存器映像从栈中恢复，因此，为了恢复寄存器映像，先得知道寄存器映像被保存在哪个栈中，也就是咱们得在切换前把当前的栈指针保存在某个地方，下次再被调度上处理器前，再从相同的地方恢复栈指针，将栈中的寄存器映像重新装载到处理器。为了方便编程，这个地方就选 PCB 中的成员 self_kstack。当然这是有意为之的，self_kstack 在 PCB 中的偏移量为 0，因此在后面可以直接用 PCB 的地址作为保存 esp。

您看，switch_to 和 PCB 是配合工作的，在 switch_to 中 self_kstack 已被固定引用为偏移 PCB 0 字节的地方，因此必须要把 self_kstack 放在 PCB 的起始处，即 task_struct 的开头。

第 11 行的“mov eax, [esp + 20]”，结合图 9-13 可发现，其中“[esp + 20]”是为了获取栈中 cur 的值，也就是当前线程的 PCB 地址，再将它 mov 到寄存器 eax 中，因为 self_kstack 在 PCB 中偏移为 0，所以此时 eax 可以认为是当前线程 PCB 中 self_kstack 的地址。

第 12 行 mov [eax], esp 将当前栈顶指针 esp 保存到当前线程 PCB 中的 self_kstack 成员中。

好啦，至此，当前线程的上下文环境算是保存完毕。下面要准备往处理器上装载新线程的上下文啦。

第 16 行“mov eax, [esp + 24]”，结合图 9-13，其中“[esp + 24]”是为了获取栈中的 next 的值，也就是 next 线程的 PCB 地址，之后将它 mov 到寄存器 eax，同样此时 eax 可以认为是 next 线程 PCB 中 self_kstack 的地址。因此，“[eax]”中保存的是 next 线程的栈指针。

第 17 行“mov esp, [eax]”是将 next 线程的栈指针恢复到 esp 中，经过这一步后便找到了 next 线程的栈，从而可以从栈中恢复之前保存过的寄存器映像。

接下来的第 19～22 行按照寄存器保存的逆顺序，依次从栈中弹出。

注意，不要误以为此时恢复的寄存器映像是在上面刚刚保存过的那些寄存器。在同一次 switch_to 的调用执行中，第 15 行之前保存的寄存器属于当前线程 cur，第 15 行之后恢复的寄存器映像属于下一个上处理器运行的线程 next，这些被恢复的寄存器映像是在之前某次执行 switch_to 时，由现在的这个 next 线程作为那时候的当前线程 cur，被换下处理器前保存的，也是用前 15 行代码保存的，只是它们分属于不同的 switch_to 调用次序。

第 23 行的 ret 便将当前栈顶处的值作为返回地址加载到处理器的 eip 寄存器中，从而使 next 线程的代码恢复执行。

如果此时的 next 线程之前尚未执行过，马上开始的是第一次执行，此时栈顶的值是函数 kernel_thread 的地址，这是由 thread_create 函数设置的，执行 ret 指令后处理器将去执行函数 kernel_thread。如果 next 之前已经执行过了，这次是再次将其调度到处理器的话，此时栈顶的值是由调用函数 switch_to 的主调函数 schedule 留下的，这会继续执行 schedule 后面的流程。而 switch_to 是 schedule 最后一句代码，因此执行流程马上回到 schedule 的调用者 intr_timer_handler 中。schedule 同样也是 intr_timer_handler 中最后一句代码，因此会完成 intr_timer_handler，回到 kernel.S 中的 jmp intr_exit，从而恢复任务的全部寄存器映像，之后通过 iretd 指令退出中断，任务被完全彻底地恢复。

switch_to 到这就结束了，它略微有些难懂，这主要体现在。

（1）switch_to 的操作对象是线程栈 struct thread_stack，对栈中的返回地址及参数的设置可能会感觉有点糊涂。因此建议别只看局部，从全局上看 kernel.S、interrupt.c、timer.c、thread.c，它们之间是密切配合的。

（2）上下文的保护工作分为两部分，第一部分用于恢复中断前的状态，这相对好理解。咱们的函数 switch_to 完成的是第二部分，用于任务切换后恢复执行中断处理程序中的后续代码。

小弟虽然已经尽力做到掰开了、揉碎了的解释，但如果不跟一下代码的话还是不容易理解，下一节咱们得上机测试一下了。

4．启用线程调度

前面的基础部分完成得差不多了，为了让线程调度跑起来，现在还要修改几个地方。先看代码 9-13，在 thread.c 中加入线程相关信息的初始化。

代码 9-13　（project/c9/c/ thread/thread.c）

```
…略
128 /* 初始化线程环境 */
129 void thread_init(void) {
130    put_str("thread_init start\n");
131    list_init(&thread_ready_list);
132    list_init(&thread_all_list);
133 /* 将当前 main 函数创建为线程 */
134    make_main_thread();
135    put_str("thread_init done\n");
136 }
137
```

在代码 9-13 中就做了三件事，通过 list_init 函数将就绪队列 thread_ready_list 和全部队列 thread_all_list 初始化，初始化的内容就是将队列为空，也就是使队列首尾相接。

再通过函数 make_main_thread 将当前已运行的主函数封装为线程，本质上就是在其 PCB 中写入了线程信息。

好啦，记得把 thread_init 加入到 init.c 中，这个就一句话的事，不用再贴代码了，大伙儿看代码实例吧。

咱们看看如何在 main.c 中创建多个线程吧。请看代码 9-14。

代码 9-14　（project/c9/c/kernel/main.c）

```
…略
 6 void k_thread_a(void*);
 7 void k_thread_b(void*);
 8 int main(void) {
 9     put_str("I am kernel\n");
10     init_all();
```

```
11
12     thread_start("k_thread_a", 31, k_thread_a, "argA ");
13     thread_start("k_thread_b", 8, k_thread_b, "argB ");
14
15     intr_enable();// 打开中断，使时钟中断起作用
16     while(1) {
17        put_str("Main ");
18     };
19     return 0;
20 }
21
22 /* 在线程中运行的函数 */
23 void k_thread_a(void* arg) {
24 /* 用 void*来通用表示参数，
   被调用的函数知道自己需要什么类型的参数，自己转换再用 */
25     char* para = arg;
26     while(1) {
27        put_str(para);
28     }
29 }
30
31 /* 在线程中运行的函数 */
32 void k_thread_b(void* arg) {
33 /* 用 void*来通用表示参数，
   被调用的函数知道自己需要什么类型的参数，自己转换再用 */
34     char* para = arg;
35     while(1) {
36        put_str(para);
37     }
38 }
```

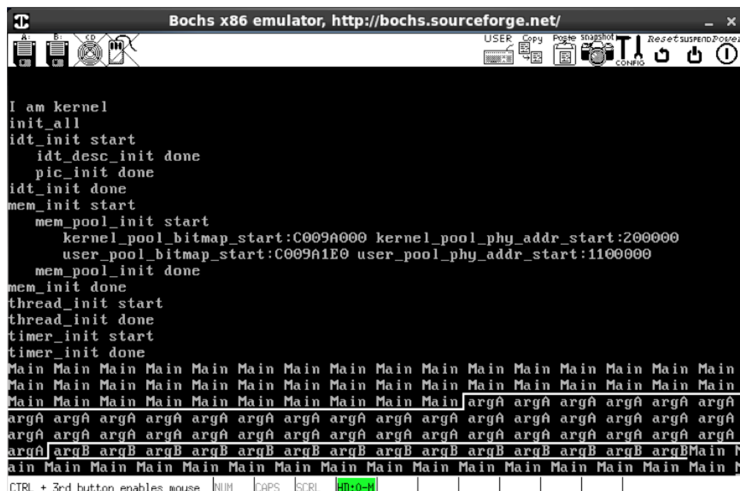
main.c 中，增加了一个函数 k_thread_b，它同 k_thread_a 是一样的，循环打印参数 arg。

我们在第 11～12 行通过 thread_start 函数把 k_thread_a 和 k_thread_b 封装为线程，传入的参数分别是“argA ”和“argB ”，注意，参数字符串结尾处有个空格。另外，我们给 k_thread_a 的优先级为 31，k_thread_b 的优先级为 8，按理说，屏幕上打印的字符串“argA ”的数量大约为“argB ”的 4 倍，一会儿我们看看是不是这样。

第 15 行通过 intr_enable 将中断打开，目前我们在 8259A 中只打开了时钟中断，因此，时钟中断对应的中断处理程序会引发调度。

另外，我们已经将 main 函数在 thread_init 中通过 make_main_thread 封装为线程，其优先级为 31，因此 main 中第 17 行的循环打印"Main "也会不断被调度。

好啦，大家自行更新 makefile，我就直接上效果图啦，如图 9-15 所示。



▲图 9-15 不同优先级线程调度

您看，三个线程分别打印了自己的参数，字符串“Main ”和“argA ”的数量差不多，因为它们的优

先级相同。线程 k_thread_b 的优先级是它们的四分之一，因此“arg_B”的数量大概也是它们的四分之一左右。

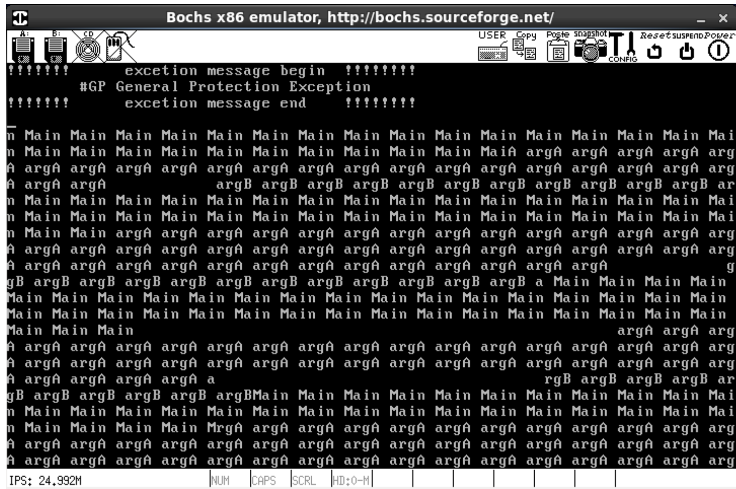
到这似乎可以告一段落了，但实际上我们还有一个很大的隐患，咱们下一节再说。

5．一般保护性异常的调试

上一节中，咱们似乎已经大功告成了，但眼力好的同学可能注意到了一个问题，图 9-15 中，最后一个“argB”和下一个“Main”之间似乎不正常，它们之间少了空格。怎么回事？

其实这可能不是问题，因为字符串中的字符是逐个打印的，有可能尚未打印完字符串中的所有字符便被换下处理器了。不过也可能真是问题，在目前情况下存在“少字符”的情况，一会儿解释。

其实我和大家隐瞒了一件事，如果再继续执行的话，就出现问题了，我把事故的现场给大伙儿贴个图看看，如图 9-16 所示。



▲图 9-16　GP 异常

大家看，图 9-16 是一幅“支离破碎、惨不忍睹”的情景，除了屏幕中有大片连续的异常空格外，更严重的问题还在后面呢，程序在屏幕最上方报出了“#GP General Protection Exception”，即一般保护性异常，这是为什么呢？

如果您学过操作系统并且已经知道了原因，还是先配合一下，先假装不知道，或者跳过相关章节。

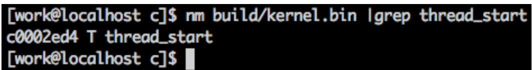
程序能打印这个报错，是因为咱们提前把默认的中断处理函数 general_intr_handler 改为了打印报错信息，因此，突然它派上用场了，难过之余，我还是略感一些幸运的。

用之前介绍过的方法，咱们在 bochs 中调试一下，看看到底是怎么回事。这里我就不详细解释每一步啦，毕竟之前咱们有过详细的过程演示。

报错之前肯定是因为某些指令导致了 GP 异常，因此我们要获取此异常发生的时间点。咱们是利用 show int 命令来配合的，这样当程序中有中断发生时就会在控制台上输出信息，无论是硬中断、软中断或是 iret 返回，通通都会显示出来。不过到目前为止，咱们还没有软中断，因此直接执行 show extint 比较直接。但如果从程序开始时就执行了 show extint，这会严重拖慢执行的速度，我们尽量在离发生 GP 异常近一点的地方再执行 show extint。因此，我们先选好一个离异常发生较近的地址作为断点。

尽管我们可以在内核文件的加载地址 0xc0001500 停下来，但为了顺便多演示一些命令的用法，我还是不惜麻烦一些。咱们不是在 main.c 中调用 thread_start 函数创建了线程吗？那好，咱们就在它的地址处打个断点，为了获取它的地址，这里用到了 nm 命令，如图 9-17 所示。

好，我们已经知道 thread_start 的虚拟地址是 0xc0002ed4。那我们用此地址在 bochs 中通过 lb 命令设置断点，如图 9-18 所示。



▲图 9-17　nm 命令查看符号地址

```
<bochs:1> lb 0xc0002ed4
<bochs:2> c
(0) Breakpoint 1, 0xc0002ed4 in ?? ()
Next at t=18111683
(0) [0x000000002ed4] 0008:c0002ed4 (unk. ctxt): push ebp                 ; 55
<bochs:3> info b
Num Type           Disp Enb Address
  1 lbreakpoint    keep y   0xc0002ed4
<bochs:4> d 1
<bochs:5> info b
Num Type           Disp Enb Address
<bochs:6> show extint
show external interrupts: ON
show mask is: extint
<bochs:7> c
    ...
    略
    ...
00019811349: exception (not softint) 0008:c0001f39 (0xc0001f39)
00019811869: exception (not softint) 0008:c0001f39 (0xc0001f39)
00019812389: exception (not softint) 0008:c0001f39 (0xc0001f39)
00019812909: exception (not softint) 0008:c0001f39 (0xc0001f39)
00019813429: exception (not softint) 0008:c0001f39 (0xc0001f39)
00019813949: exception (not softint) 0008:c0001f39 (0xc0001f39)
00019814327: exception (not softint) 0008:c0001d2b (0xc0001d2b)
^CNext at t=134183107
(0) [0x000000001818] 0008:c0001818 (unk. ctxt): jmp .-2 (0xc0001818)      ; ebfe
<bochs:8>
```

▲图 9-18　调试过程

在键入命令 c 后，程序一直执行到断点处。由于 thread_start 我们在 main.c 中调用了两次，所以程序会再次在相同地址处停下来，因此先将此断点删除。

通过 info b 命令查看当前的断点，就这一个，因此其 num 为 1。将此 num 作为断点删除命令 d 的参数，在执行命令 d 1 后该断点就被删除了。再次执行 info b 命令，断点结果集为空。

是时候执行 show extint 了，之后控制台上会源源不断打印中断信息，太长了，因此我略过了一些。直到控制台上不再输出中断信息时，这说明程序此时已经进入了 general_intr_handler 函数中的 while(1)死循环，这时屏幕上已经开始打印“#GP General Protection Exception”，这与咱们的判断相符。

我们的目地是为了获取异常发生的时间点，我已经在图 9-18 中用下画线标出来了，值为 19814327，这表示 GP 异常发生的时间，此时间点前的指令必然是导致异常的原因，因此我们将此值减 1，用 19814326 作为时间点断点。继续看图 9-19。

```
<bochs:1> sba 19814326
Time breakpoint inserted. Delta = 19814326
<bochs:2> c
(0) Caught time breakpoint
Next at t=19814326
(0) [0x000000001fd0] 0008:c0001fd0 (unk. ctxt): mov byte ptr gs:[bx], cl  ; 6567
880f
<bochs:3> r
eax: 0xc000cfcf -1073688625
ecx: 0x00000020 32
edx: 0xc00003d5 -1073740843
ebx: 0xc0009f9e -1073700962
esp: 0xc0101de4 -1072685596
ebp: 0xc0101e40 -1072685504
esi: 0x00000000 0
edi: 0x00000000 0
eip: 0xc0001fd0
eflags 0x00000283: id vip vif ac vm rf nt IOPL=0 of df IF tf SF zf af pf CF
<bochs:4> info gdt
Global Descriptor Table (base=0xc0000900, limit=31):
GDT[0x00]=??? descriptor hi=0x00000000, lo=0x00000000
GDT[0x01]=Code segment, base=0x00000000, limit=0xffffffff, Execute-Only, Non-Con
forming, Accessed, 32-bit
GDT[0x02]=Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed
GDT[0x03]=Data segment, base=0xc00b8000, limit=0x00007fff, Read/Write, Accessed
You can list individual entries with 'info gdt [NUM]' or groups with 'info gdt [
NUM] [NUM]'
<bochs:5> x gs:bx
WARNING: Offset 00009F9E is out of selector 0018 limit (00000000...00007fff)!
[bochs]:
0xc00c1f9e <bogus+       0>:    0x18306000
<bochs:6> s
Next at t=19814327
(0) [0x000000001d2b] 0008:c0001d2b (unk. ctxt): nop                       ; 90
<bochs:7> s
Next at t=19814328
(0) [0x000000001d2c] 0008:c0001d2c (unk. ctxt): push ds                   ; 1e
```

▲图 9-19　调试过程（续）

插入时间点断点的命令是 sba，我们执行了 sba 19814326 后，键入命令 c 持续执行，直到断点处停下来。

马上要执行的命令是"mov byte ptr gs:[bx],cl"，图中第一个下画线标出的就是。段寄存器 gs 用来存储显存段的选择子，所以这是要往显存段写数据，也就是咱们打印功能的底层实现。mov 命令仅仅涉及到寄存器和内存的读写，寄存器的操作基本不会出错，因此很可能是访问内存时出了问题，这里的内存寻址用到了寄存器 ebx 的值，先不急，咱们先通过 r 命令查看通用寄存器。您看，其中 ebx 的值为 0xc0009f9e，也就是 bx 的值为 0x9f9e，哦！如果您熟悉显存段描述符，似乎找到问题了。

回忆一下，文本模式下显存物理地址范围为 0xb8000～0xbffff，咱们的显存段描述符是按照此范围来设置的，因此段基址为 0xb8000，段大小为 0xcffff-0xb8000=0x7fff，寄存器 bx 用作显存段内的偏移量，因此它不应该超过 0x7fff，但它的值是 0x9f9e，明显已经越界了。更何况，咱们的显存操作范围仅限于 4000 字节之内的显存，也就是显存范围是 0xb8000～0xb8fa0，因此不会用到 0xb8fa1～0xbffff 的范围，更不可能超过 0xbffff，原因是由于咱们的滚屏操作中有相关的判断，当超过 2000 个字时就会滚屏了，这样做的初衷是图简单省事，否则显卡操作可不会这么简单。

根据图 9-19，用 info gdt 命令输出了安装的段描述符信息，其中 GDT[0x03]便是显存段描述符，其段基址 base 为 0xc00b8000，段界限 limit 为 0x00007fff，因此实际的显存段范围为 0xc00b8000～ (0xc00b8000+0x7fff)，即 0xc00b8000～0xc00bffff。

寄存器 bx 的值 0x9f9e 明显已经超过了段界限 limit 的值 0x7fff，在特权检查时自然无法通过，所以才会报 GP 异常。

咱们用查看内存的命令 x 试探一下该地址便知，如果该地址可以被正常访问的话，说明该地址通过了 CPU 的各种检测。您看，在执行了 x gs:bx 后，控制台上果然输出了 warning，提示超过了选择子 0x18 对应的段的 limit。

紧接着我用单步执行命令 s 执行了这条引发异常的指令，果然进入了中断，nop 便是充当异常的 error_code 位，下面的指令 push ds 又进一步验证了这一想法，这是 kernel.S 中进入中断时的上下文保护代码。

导致异常的表面原因找到了，但根本原因目前还是不知道，上面寄存器 bx 的值 0x9f9e 是哪来的呢？

其实这个问题还是有点复杂的，我们还是得专门用一个章节讲它，我们还是在下一章的同步机制中讲解。

第 10 章　输入输出系统

本章将实现简单的输入输出，它是我们将来能通过 shell 命令同系统交互的基础。话说我们用户进程还没实现呢，shell 交互是很久以后的事。

10.1 同步机制——锁

上一章中我们遇到的字符混乱和 GP 异常问题，根本原因是由于临界区代码的资源竞争，这需要一些互斥的方法来保证操作的原子性。其实学过操作系统课程的读者早就提前知道了答案。

10.1.1　排查 GP 异常，理解原子操作

上一节中，咱们重点考察的是引起 GP 异常的原因是寄存器 bx 的值 0x9f9e 超过了显存段的范围，从而在特权检查时引起了安全保护，这只是表面的现象，我们依然无法解决问题。因此咱们必须得清楚此值是怎么来的，知道了这个原因后才能从根本上把问题解决。

咱们先分析下待解决的几个问题。

（1）图 9-15 的输出中，有些字符串看似少了字符。

（2）图 9-16 的输出中，有大片连续的空缺。

（3）图 9-16 的 GP 异常。

前两个其实是有“共性”的，都是字符打印混乱的问题，第 3 个问题看似“较独立”，但解决起来似乎难度很大，要不咱们先从简单的问题入手，看看为什么字符会打印混乱。

拿图 9-15 为例，输出的字符串可分为三类，“Main”“argA”“argB”，在每一类连续输出的字符串中，同一类的内部是正常的，字符串整洁有序，不多不少，少字符的部分只是出现在不同种类的字符串“交界”的地方。

这说明什么？和线程调度有关？

线程调度工作的核心内容就是线程的上下文保护+上下文恢复。上下文恢复是上下文保护的逆操作，如果保护部分没问题，恢复也会被排除掉“嫌疑”。咱们先从保护工作入手。

上下文包括两部分，第一部分用于中断时的保护，第二部分用于调度时的保护。

是否是上下文保护的第一部分出了问题？下一个待打印的字符地址给搞错了？

答：在每一类字符串连续输出的过程中，时钟中断都在持续打断任务的执行，否则线程是不会被换下处理器的。拿主线程为例，其优先级为 31，每一次时钟中断都会将当前线程的 ticks 减 1，因此，图 9-15 中在首次连续输出 42 个“Main ”期间，中断发生了 31 次，这说明个别的字符串“Main ”并不是连续打印的，而是中间插入了时钟中断处理程序，即先打印了前几个字符，被中断断开，恢复后回来又继续打印后几个字符。而我们看到，同类的字符串内部是连续完整输出的，说明线程在自己的时间片内工作正常，第一部分的上下文保护也是正确的。

字符串丢失发生在字符串交界处，也就是说当调度了新的线程后才出现了丢字符的现象。难道是线程上下文保护的第二部分出了问题？

答：这个不会有问题，因为第二部分的上下文保护是为了保护线程在内核调度程序中的上下文，即 switch_to 前后的寄存器环境，这与线程待打印的字符串无关，和字符串打印有关的寄存器映像是由第一部分保存的，第二部分的作用仅仅是保证线程被 switch_to 恢复后能够顺利执行到出口 intr_exit，

在 intr_exit 处的代码是用第一部分保存的寄存器映像恢复线程上下文，从而退出中断。而它我们已经讨论过了，应该不会出问题。况且在图 9-15 中，第一组连续的“Main ”和第一组连续的“argA ”的交界处是正常的，它并没有出现丢字符的现象。

上下文保护看来是正确的，似乎线程调度不是问题所在。字符可以打印，这似乎说明字符打印的功能并没有问题，问题变得有意思了，到底哪里出了问题呢？

回忆一下，线程创建之前，在 init_all 函数中咱们也调用了 put_str 函数，在各模块的初始化中也有 put_str 的调用，那时候的字符串打印是正确的，与现在的区别是那时候是在关中断的情况下，现在是在开中断的情况下。难道和中断有关？可是，之前已经验证过，图 9-15 中那连续的 42 个“Main ”也是在开中断的情况下打印的，这……虽然有点凌乱，但咱们还是看出了另一些不同，之前的“正常”是在只有一个主线程在关中断下，现在的“异常”是多线程在开中断下，难道是开中断时多线程才会出问题吗？

越到没头绪的时候，越要从基础入手，也许大部分同学都在建议：“不要创建那两个线程了，先只用一个主线程试试看”。好主意，就听大伙儿的，咱们先看看单线程开中断下是否会出问题。我在 main.c 中把第 12～13 行的 thread_start 注释后，编译运行，效果如图 10-1 所示。



▲图 10-1　单线程字符串打印效果

哇，当只创建一个线程，即主线程自己被调度来调度去的时候，字符串打印显得无比和谐，井然有序。看来有希望了，似乎是多线程开中断下才会出问题。

那会不会是多个线程在打印字符时，出现互相覆盖的现象呢？看来现在必须得把心踏实下来，咱们从最基础的打印流程入手。

回忆一下，字符打印的核心函数是 put_char，它的功能是在光标处打印 1 个字符。让我们分析下字符打印函数 put_char 的工作原理。

1．获取光标值

在咱们的逻辑中，光标值是下一个打印字符的地址。为了知道在哪个位置打印字符，先得通过读取显卡的光标寄存器获取光标值。

2．将光标值转换为字节地址，在该地址处写入字符

光标值是字符的次序，一个字符占显存中的 2 字节，因此要将光标值乘以 2，将光标转换为字符在显存中的偏移量，此偏移量+显存的起始地址 0xb8000 所得的地址和，便是下一个字符在显存中的地址，随后在该地址处写入字符 ASCII 码及属性。

由于在步骤 2 中又要判断字符类型，又要写入字符，又要判断是否滚屏，所以步骤 2 中的微步骤很多。

3．更新光标的值

字符写入完成后，将显存中下一个可以打印字符的地址转换为光标值后写入显卡的光标寄存器，供下

一次字符打印时读取。

以上就是打印一个字符的三个步骤。按理来说，这三个步骤必须一气呵成，要么全都完成，要么一个步骤都不做，总之打印字符这件事不可拆分，更确切地说，不能被调度机制拆分，说白了，就是要求字符打印执行过程中不能被切换成其他任务。字符打印过程中的三个步骤像原子一样不可拆分，因此字符打印必须具有原子性。

您肯定想到了，每个任务都有时间片限制，迟早会执行任务调度，所以，任务调度保不准就是在某个线程执行字符打印时发生的，也就是说，字符打印中的三个步骤被拆开了。线程在时间片内执行时并不会受调度器的影响，因此，此过程中字符打印“算是”具有原子性，在任务调度前的执行期间，打印的是连续完整的字符串。在任务调度前的一刻，如果此线程执行了 put_char 中的步骤 1，并且步骤 3 尚未执行（不管第 2 步是否执行完），那么就一定会出问题。

那现在可以解释“少字符”的现象了。大家都知道，各个线程都是调用 put_str 函数来打印字符串的，而在 put_str 内部是对字符串中的各个字符分别调用 put_char 来逐个打印的。

下面我们把现场还原，注意，这里要访问的公共资源是显存。

假设线程 k_thread_a 正在通过 put_char 打印某个字符，它在 put_char 中刚刚完成了字符打印的步骤 1，即读取了字符打印的光标位置，假设获取到的光标值为 C1。于是它就兴高采烈地进行步骤 2，也就是“打算”将光标值转换成地址后，再往该地址处写入字符的 ASCII 码及属性。由于步骤 2 中的微步骤很多，所以步骤 2 的完成需要一步一步来，不管步骤 2 是否执行彻底，总之尚未进行步骤 3，也就是还没来得及往光标寄存器中写入新的光标值时，此时时钟中断发生了，并且线程 k_thread_a 的时间片到期，它被调度程序 schedule 换下了处理器，将线程 k_thread_b 换上处理器。k_thread_b 运行后，也是间接通过 put_char 打印字符，它开始执行步骤 1 获取光标值，线程 k_thread_a 之前未更新光标值，k_thread_b 此时获取的光标值也是 C1。所以会在相同的地方打印字符，也就是字符覆盖了，这就是“少字符”的原因。在任务调度时才会出现这种情况，因此字符覆盖一定出现在不同组字符串的交界处。

也许有读者想到了，既然是任务调度破坏了字符打印的原子性，而任务调度又是由时钟中断调用的，是否可以在字符串打印前后通过关中断的方式来保证原子性？我想可以，要不咱们试试，不过为了省事，咱们就不把关中断在 put_char 的前后开关中断了，放在 put_str 前后吧，请看代码 10-1。

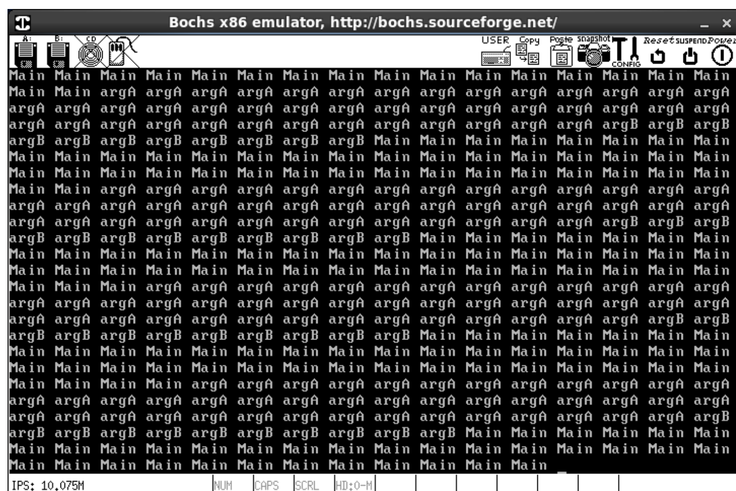
代码 10-1　（project/c10/a/kernel/main.c）

```
…略
16   while(1) {
17        intr_disable();       // 关中断
18        put_str("Main ");
19        intr_enable();        // 开中断
20   };
21   return 0;
22 }
23
24 /* 在线程中运行的函数 */
25 void k_thread_a(void* arg) {
26 /* 用 void*来通用表示参数，
   被调用的函数知道自己需要什么类型的参数，自己转换再用 */
27   char* para = arg;
28   while(1) {
29       intr_disable();        // 关中断
30       put_str(para);
31       intr_enable();         // 开中断
32   }
33 }
34
35 /* 在线程中运行的函数 */
36 void k_thread_b(void* arg) {
37 /* 用 void*来通用表示参数，
   被调用的函数知道自己需要什么类型的参数，自己转换再用 */
38   char* para = arg;
39   while(1) {
40       intr_disable();        // 关中断
41       put_str(para);
42       intr_enable();         // 开中断
```

```
43     }
44 }
```

在打印前先把中断关闭，在打印完成后，还是要把中断打开的，否则无法调度新线程，因此在 put_str 前后都加了 intr_disable 和 intr_enable。

编译运行，效果如图 10-2 所示。



▲图 10-2　打印前后关中断

图 10-2 所示是运行结果的快照，在持续运行了 10 分钟后，看上去一切良好。顺便说一句，每个字符串之所以都能“完整地”打印，即原字符串中的每个字符顺序挨着，是因为关、开中断操作放在了 put_str 的前后，如果放在了 put_char 的前后，各字符串组的两端也可能是混乱的。

现在该讨论下最严重的错误了，那个导致 GP 异常的 0x9f9e 是从哪来的呢？

考虑一下现在的情况。

（1）大家想想看，我们在函数 put_char 中明明有对光标值的判断，即当下一个写入字符的光标位置大于等于 2000 时就会滚屏，滚屏后会把光标置为屏幕最后一行的行首光标值，即 1920，也就是说我们所写入光标寄存器的光标值，应该始终小于 2000，即 0～1999 的范围。

（2）图 9-19 中，第一个下画线的代码“mov byte ptr gs:[bx],cl”是导致异常的指令，其中 bx 的值是刚刚从光标寄存器读进来的，因此可以判断，一定是某个线程在上一次往光标寄存器中更新了错误的光标值。

情况（1）是说所设置的光标一定小于 2000，情况（2）是说所设置的光标一定大于等于 2000，看似矛盾啊，好凌乱地说……

其实，情况（1）是种“愿景”，或者说是原子操作下的理想态：从光标读取，到写入字符，最后到更新光标都应该属于同一个线程的工作，也就是说这三个操作不应该由多个线程重复“掺和”……似乎有点眉目了。

先看一下为什么寄存器 bx 的值会变成 0x9f9e 呢？

首先，看过代码的同学肯定知道，导致异常的语句“mov [gs:bx], cl”位于函数 put_char 中的标号.put_other 后面，其中寄存器 bx 用作显存操作中的偏移量。此偏移量是由 shl bx, 1 语句将光标值乘以 2 得来的，因此，有必要先将 bx 中的地址 0x9f9e 还原为光标值，看看原本读进来的光标值到底是多少。

咱们从后往前推，将 bx 中的地址恢复为光标。不知道大伙儿注意到没有，其中 eflags 寄存器的 CF 位为 1（用方框框起来的 CF），这表示有进位，源代码中“shl bx, 1”和“mov gs:[bx],cl”是挨着的，因此 CF 位表示经过“shl bx, 1”左移 1 位（乘以 2）后有进位，这说明 bx 的值虽是 0x9f9e，但它只是部分结果，bx 是 16 位操作数，由 shl bx,1 把结果的进位抹掉了，即结果应该为 0x19f9e。将此值除以 2 后的结果为 0xcfcf，也就是读进来的光标值为 0xcfcf。

其实在 put_char 开头部分，光标最初是存入到寄存器 ax 中的，bx 中的值是由指令 mov bx, ax 从 ax 复制过去的，图 9-19 中，用 r 命令显示的寄存器中，eax 的值为 0xc0009f9e，因此光标初值再次被证实为 0x9f9e。

顺便提一下，此时 ebx 的值虽为 0xc0009f9e，但高 2 字节的 0xc000 并不是 put_char 中用到的，它是由主调函数 put_str 记录字符串地址用的，因此高 2 字节以 0xc000 开头。我们的显存段基址是 0xc00b8000，因此我们此处只用 bx 来做显存的偏移地址。

另外，既然咱们的光标值范围是 0～1999，将其转换为十六进制的话，即 0～0x7cf，这说明光标值最多为 3 位十六进制数，并且高字节始终小于等于 7。

而我们读进来的光标值为 0xcfcf，即高字节为 2 个十六进制位，因此一种可能设置错误光标值的情况是：光标值的低字节被当成高字节写入到光标寄存器中了，也就是问题出在 put_char 结尾处的光标设置阶段，而且这必然是由两个线程共同“掺和”办的坏事，下面讨论下这种情况。

在咱们的代码中，光标设置分为 4 个微操作。

（1）通知光标寄存器要设置高 8 位，如图 10-3 所示。

（2）输入光标值的高 8 位，如图 10-4 所示。

（3）通知光标寄存器要设置低 8 位，如图 10-5 所示。

（4）输入光标值的低 8 位，如图 10-6 所示。

```
;;;;;;; 1 先设置高8位 ;;;;;;;;
   mov dx, 0x03d4
   mov al, 0x0e
   out dx, al
```

▲图 10-3

```
   mov dx, 0x03d5
   mov al, bh
   out dx, al
```

▲图 10-4

```
;;;;;;; 2 再设置低8位 ;;;;;;;;;
   mov dx, 0x03d4
   mov al, 0x0f
   out dx, al
```

▲图 10-5

```
   mov dx, 0x03d5
   mov al, bl
   out dx, al
```

▲图 10-6

前两个步骤（1）和（2）是为了设置光标的高 8 位，后两个步骤（3）和（4）是为了设置光标的低 8 位。经过完整的 4 个步骤，新的 16 位光标值才被写入光标寄存器。

虽然我们本次实例中有 3 个线程，但以上 GP 异常的情况在 2 个线程的环境中也会出现（测试过），因此，为叙述方便，下面咱们以两个线程为例，根据以上 4 个步骤来复现 GP 异常的现场，注意，这里要访问的公共资源是光标寄存器（都是公共资源惹的祸）。

假设线程 k_thread_a 正在处理器上运行，不管它运行了多久，在此次时间片内的最后一刻，恰好它去设置光标，此时 k_thread_a 要设置的光标值为 0x7cf，即 1999，也就是下一个字符的位置是屏幕右下角。在完成上面第（3）步的微操作后，第（4）步尚未开始，此时发生了中断，由于此线程的时间片耗尽，调度器把 k_thread_a 换下处理器，调度 k_thread_b 运行。大伙儿注意，目前光标寄存器中仅更新了高 8 位（虽然其值为 0x7，但它不重要），低 8 位的设置未彻底。光标寄存器的值仍为 0x7ce，线程 k_thread_a 只是把光标寄存器的高 8 位更新为 0x7，当它下次再运行时，首先做的就是把第 4 步完成，即把 0xcf 写入光标寄存器的低 8 位。

k_thread_b 运行后，它先获取了光标值 0x7ce，然后将光标转换成显存的偏移地址，随后在该地址处输出字符。当输出一个字符后，光标值被更新为 0x7cf，这样在输出下一个字符时，获取到的光标值便为 0x7cf，接着在此光标值对应的显存地址处写入字符。随后将寄存器 bx 自加 1，它作为下一个光标值，此时 bx 的值为 2000，这时候需要滚屏了。

滚屏操作相对来说比较繁琐，而且线程 k_thread_b 的时间片很少，故大部分时间都被消耗在滚屏上了。在它终于完成滚屏后，下一步就是去更新光标值。当它刚刚完成第 1 步，第 2 步尚未开始的时候，恰好又被中断。注意，对于设置光标寄存器的高 8 位来说只完成了一半步骤，还差光标具体值没有写入，因此光标寄存器高 8 位的值还没来得及更新，依然为 0x7，整个 16 位光标值依然为 0x7cf。之前滚屏操作耗尽了时间片，因此调度器把线程 k_thread_b 换下处理器，再次换上线程 k_thread_a 运行。注意，线程 k_thread_b 刚刚执行完第 1 步，即告诉了显卡马上要写入光标值到光标寄存器的高 8 位，因此再被写入的任何 8 位光标值将被认为是存入光标寄存器的高 8 位中。

k_thread_a 运行后，继续执行自己上一步未完成的操作，也就是完成第 4 步，往光标寄存器中写入低 8 位的值，也就是写入 0xcf。但 k_thread_b 已经告诉光标寄存器要设置高 8 位，目前只差输入高 8 位的具体值了，光标寄存器把此次 k_thread_a 输入的低 8 位值 0xcf 当成了光标的高 8 位，存入了光标寄存器的高 8 位，因此，光标被更新为 0xcfcf。当下一次输出字符时，获取到的光标便为 0xcfcf，从而导致 GP 异常。

过程就是这样，有兴趣的话大伙儿可以自行调试跟踪。不过这种有竞争条件的调试是相当麻烦的，异

常只是在某些情况下才会出现，而且异常的发生取决于中断发生的时机，而我们又是在虚拟机中调试的，中断又取决于虚拟机的中断模拟策略。

总结一下，有关以上的两个问题，根本原因是访问公共资源需要多个操作，而这多个操作的执行过程不具备原子性，它被任务调度器断开了，从而让其他线程有机会破坏显存和光标寄存器这两类公共资源的现场。

至此咱们都搞清楚了，现在说说正事吧。

虽然我们通过关中断的方式暂时解决了问题，难道我们以后也要在打印前关中断吗？虽然我们可以再写一个新函数将 intr_disable、put_str 和 intr_enable 封装到一块，以后直接调用新函数就行，但以后万一还有其他情况也需要原子操作呢，难道每个都封装一遍？这种做法显然不科学，看来咱们得另辟蹊径了。

难道多个线程访问公共资源就那么可怕吗？没有办法保证它们的操作时序吗？您肯定想到了，对，正如本节标题，咱们要实现锁，相关内容咱们下一节再讨论。

10.1.2　找出代码中的临界区、互斥、竞争条件

咱们当初学习操作系统的时候，老师有讲过临界区、互斥、竞争条件、原子性、信号量和 P、V 操作等概念，其中最核心的就是临界区和互斥。其实咱们对它们一点都不陌生，因为在上节的调试过程中已经反复熟悉了它，只是我们还没介绍过它们的学名。

先给大伙介绍下这些术语，看完之后您就知道其实咱们早就理解它们了。要介绍的这些概念，本质上都是围绕着多个任务如何访问公共资源，因此，为解释清楚概念，必须在这样的前提下，即假设多个任务共享同一套公共资源。

- 公共资源

可以是公共内存、公共文件、公共硬件等，总之是被所有任务共享的一套资源。

- 临界区

程序要想使用某些资源，必然通过一些指令去访问这些资源，若多个任务都访问同一公共资源，那么各任务中访问公共资源的指令代码组成的区域就称为临界区。怕有同学看得不仔细，强调一下，临界区是指程序中那些访问公共资源的指令代码，即临界区是指令，并不是受访的静态公共资源。

- 互斥

互斥也可称为排他，是指某一时刻公共资源只能被 1 个任务独享，即不允许多个任务同时出现在自己的临界区中。公共资源在任意时刻只能被一个任务访问，即只能有一个任务在自己的临界区中执行，其他任务想访问公共资源时，必须等待当前公共资源的访问者完全执行完他自己的临界区代码后（使用完资源后）再开始访问。

- 竞争条件

竞争条件是指多个任务以非互斥的方式同时进入临界区，大家对公共资源的访问是以竞争的方式并行进行的，因此公共资源的最终状态依赖于这些任务的临界区中的微操作执行次序。

当多个任务“同时”读写公共资源时，也就是多个任务“同时”执行它们各自临界区中的代码时，它们以混杂并行的方式访问同一资源，因此后面任务会将前一任务的结果覆盖，最终公共资源的结果取决于所有任务的执行时序。这里所说的“同时”也可以指多任务伪并行，总之是指一个任务在自己的临界区中读写公共资源，还没来得及出来（彻底执行完临界区所有代码），另一个任务也进入了它自己的临界区去访问同一资源。

好啦，以上的概念基本上是为了实现原子操作而提出的，现在把这些概念在咱们的代码中对号入座，您会发现，当初操作系统课上的那些“不明觉厉”的内容原来不如过此。

咱们也先从多个任务访问公共资源的“前提”说起，大伙儿回忆一下，我们之前为什么会出现 GP 异常？原因是多个线程并行混杂访问了公共的资源，也就是说，这种访问方式不满足互斥，从而产生了竞争条件。这里的多个任务是指咱们的线程 k_thread_a、k_thread_b 和主线程。公共资源就是显卡的光标寄存器和显存。

咱们代码中的临界区是指什么？创建了三个线程，每个线程都调用 put_char 函数来打印字符，put_char 的功能是访问公共资源显存及光标寄存器，虽然 put_char 在内存中的实例就一份，但所有线程都调用了它，所以 put_char 函数属于各线程的一部分，因此它就是各线程自己的临界区。

下面看看咱们代码中有哪些“竞争条件”。

上一节中给大伙儿还原了两个现场，一个“可能是问题”的“少字符”现象，它是由于对公共资源“显存”未实现互斥访问造成的。另一个就是 GP 异常，它是对公共资源“光标寄存器”未实现互斥访问造成的。这两个现场过程就是竞争条件。处理器执行指令过程中，即使发生了中断，也会先把当前的指令执行完再处理中断，不会出现指令只执行部分的情况。因此，单条指令的执行具有原子性。

综上所述，造成这两种问题的本质原因是临界区 put_char 中的指令不是一条，而是很多，因此对公共资源的访问无法一下子执行彻底。当多个线程都在临界区时，受访的资源是同一个，加之多个线程又是伪并行，后面进入临界区的线程必然会覆盖前面所有线程的成果，再者，即使多个线程是真并行执行，对于访问共享资源也会有个前后顺序，因此显存和光标寄存器这两个公共资源的状态取决于所有线程的访问时序。

现在看看咱们代码中的“互斥”。

其实咱们已经用了个“可依赖”的互斥手段，就是进入临界区前通过函数 intr_disable 关中断。大伙儿不要因为它的“简单粗暴”就对其不屑一顾，这可是操作系统课程上介绍过的方法呢，关中断是实现互斥最简单的方法，没有之一。我们今后实现的各种互斥手段也将以它为基础。

不过话又说回来了，虽然关中断可以实现互斥，但关中断的操作应尽量靠近临界区，这样才更高效，毕竟临界区中的代码才用于访问公共资源，而访问公共资源的时候才需要互斥、排他，各任务临界区之外的代码并不会和其他任务有冲突。关中断操作离临界区越远，多任务调度越低效，不夸张地说，若将关中断操作加在了任务执行之初，多任务并行（伪并行）系统将退化成单任务串行执行。而我们做得也不好，关中断函数 intr_disable 并没有离临界区函数 put_char 足够近，比如可以放在 put_char 前，或者最好能将关中断指令 cli 作为 put_char 函数中第一条指令。我们把它放在了调用 put_char 的 put_str 函数之前，因此各组字符串结尾处的输出是完整的。但是在 put_str 中距调用 put_char 还有一些各线程私有的操作，因此在这些私有操作的时间内无法响应中断，从而其他任务无法及时被调度，降低了系统的多任务执行效率。

我怕说得不清楚，举个通俗的例子吧，就拿咱们在家吃面条来说，每个人吃面条的过程是每个人的私有操作，锅里的面条是公共资源，到锅里盛面条的动作属于临界区。通常我们都是吃完一碗后，再去锅里盛下一碗（北方人都比较能吃面），当小明在盛面条的时候，如果大明也恰好想去盛面条，大明只能在旁边等着小明盛完后，再去盛自己的面，其实这个等待就相当于互斥，只不过这个互斥是由“文明排队的素质”来保证的。若小明素质很差，他在刚端着碗、拿着筷子吃面的时候就站在锅边上，并且不让别人上前盛面，必须得等自己连吃几碗吃饱后再让别人盛，这就耽误了别人吃饭，相当于大家一个个轮流吃饭一样了。

好啦，又到了总结的时候。

多线程访问公共资源时出问题的原因是产生了竞争条件，也就是多个任务同时出现在自己的临界区。为避免产生竞争条件，必须保证任意时刻只能有一个任务处于临界区。因此，只要保证各线程自己临界区中的所有代码都是原子操作，即临界区中的指令要么一条不做，要么一气呵成全部执行完，执行期间绝对不能被换下处理器。

其实，之所以出现竞争条件，归根结底是因为临界区中的指令太多了，如果临界区仅有一条指令的话，这本身已属于原子操作，完全不需要互斥。因此，在临界区中指令多于一条时才需要互斥。当然，临界区中很少存在只有一条指令的情况，因此我们必须提供一种互斥的机制，互斥能使临界区具有原子性，避免产生竞争条件，从而避免了多任务访问公共资源时出问题。

该说的差不多都说了，下一节咱们开始讨论锁的实现。

10.1.3　信号量

我们的锁是用信号量来实现的，因此必须得和大伙儿交待清楚信号量是咋回事。

信号量在计算机世界中也算是历史悠久了，此概念是由荷兰人 E.W.Dijkstra 在 1965 年首次提出的，它是一种程序设计构造方法。其原型来自于铁路上的信号灯，大伙儿都知道，任何时候铁轨上只能有一辆火车，否则您懂的。铁路上是如何保证这一点的呢？原来它们也有个信号量系统，火车要想进入新的轨道，它必须等到相应的信号才行。

在计算机中，信号量就是个 0 以上的整数值，当为 0 时表示已无可用信号，或者说条件不再允许，因此它表示某种信号的累积“量”，故称为信号量。

虽然信号量听上去和信号相似，像是某种通信机制，但其实它和通信无关，信号量是种同步机制。

小插曲，什么是同步？同步一般是指合作单位之间为协作完成某项工作而共同遵守的工作步调，强调的是配合时序，就像十字路口的红绿灯，只有在绿灯亮起的情况下司机才能踩油门把车往前开，这就是一种同步，绿灯不亮就开车的话容易引起交通事故，这就是在十字路口这种事故多发地带用红绿灯同步交通的目的。同步简单来说就是不能随时随意工作，工作必须在某种条件具备的情况下才能开始，工作条件具备的时间顺序就是时序。

因此，配合时序并不特指时间，但却都可表示为时间。比如 NBA 比赛，球员传球时肯定得先看哪里有队友，并且队友正看着自己以表示准备好了随时接应，这个配合时序就是指队友在某个地方准备好接应，也可以理解为队友准备好接应的时候。中场休息时，啦啦队员才会上来表演，这个配合时序就是休息时间。

当多个线程共同协作完成一件工作时，它们之间的合作必须遵守某种约定，以该约定作为合作的步调，因此也需要同步。大伙儿想想看，咱们之前出现的 GP 异常，就是因为多个线程没有同步合作导致的，要是大家都约定：凡是有线程在访问公共资源，其他线程就不要进来捣乱，自然不会出任何问题。

线程同步的目的是不管线程如何混杂、穿插地执行，都不会影响结果的正确性。线程不像人那样有判断“配合时序”的意识，它的执行会很随意，这就使合作出错成为必然。因此，当多个线程访问同一公共资源时（当然这也属于线程合作），为了保证结果正确，必然要用一套额外的机制来控制它们的工作步调，也就是使线程们同步工作。

再回来说信号量。信号量就是个计数器，它的计数值是自然数，用来记录所积累信号的数量。这里的信号是个泛指，取决于信号量的实际应用环境。可以认为是商品的剩余量、假期剩余的天数、账号上的余额等，总之，信号的意义取决于您用信号量来做什么，信号量仅仅是一种程序设计构造方法。

既然信号量是计数值，必然要有对计数增减的方法。由于 Dijkstra 是荷兰人，他用 P、V 操作来表示信号量的减、增，这两个都是荷兰语中的单词的缩写。P 是指 Proberen，表示减少，V 是指单词 Verhogen，表示增加。但计算机一直是美国最发达，而且 P、V 显得意义不明朗（由于字母 V 很像是朝下指的箭头，我经常觉得这是使信号量减少的操作，为了强制记忆，我把字母 P 的发音记成往下“劈”的动作，这才记住 P 才是减少信号量的操作），因此对信号量的加法操作是用 up 表示，减法操作是用 down 表示。下面介绍下这两种操作。

增加操作 up 包括两个微操作。

（1）将信号量的值加 1。

（2）唤醒在此信号量上等待的线程。

减少操作 down 包括三个子操作。

（1）判断信号量是否大于 0。

（2）若信号量大于 0，则将信号量减 1。

（3）若信号量等于 0，当前线程将自己阻塞，以在此信号量上等待。

信号量是个全局共享变量，up 和 down 又都是读写这个全局变量的操作，而且它们都包含一系列的子操作，因此它们必须都是原子操作。

信号量的初值代表是信号资源的累积量，也就是剩余量，若初值为 1 的话，它的取值就只能为 0 和 1，这便称为二元信号量，我们可以利用二元信号量来实现锁。

在二元信号量中，down 操作就是获得锁，up 操作就是释放锁。我们可以让线程通过锁进入临界区，可以借此保证只有一个线程可以进入临界区，从而做到互斥。大致流程为：

- 线程 A 进入临界区前先通过 down 操作获得锁（我们有强制通过锁进入临界区的手段），此时信号量的值便为 0。
- 后续线程 B 再进入临界区时也通过 down 操作获得锁，由于信号量为 0，线程 B 便在此信号量上等待，也就是相当于线程 B 进入了睡眠态。
- 当线程 A 从临界区出来后执行 up 操作释放锁，此时信号量的值重新变成 1，之后线程 A 将线程 B 唤醒。
- 线程 B 醒来后获得了锁，进入临界区。

好啦，信号量这种抽象的东西这么干说也不容易理解，咱们尽快看看具体的实现吧。

10.1.4 线程的阻塞与唤醒

其实，咱们离实现锁只有一步之遥了。

大伙儿已经知道了，咱们是用二元信号量来实现锁的，信号量 down 操作中的第 3 个微操作提到了阻塞当前线程的功能，信号量 up 操作中的第 2 个微操作提到了唤醒线程的功能，因此在实现锁之前，我们必须提前实现这两个功能。

我们用函数 thread_block 实现了线程阻塞，用函数 thread_unblock 实现了线程唤醒。

先说一下阻塞的原理。

不知道大伙儿对阻塞是否好奇，阻塞是什么？是一种阻碍线程运行的神奇力量吗？怎么就把线程阻塞了呢？反正我当初觉得好神奇。

大伙儿知道，“阻塞”就是指不能运行，或者说不让运行，这只是个概念，不同的系统有不同的实现，但实现原理都是一样的。在我们的系统中，大伙儿想想看，为什么线程可以运行呢？还不是调度器将线程从就绪队列中摘出来放到处理器上吗。哈哈，似乎一下子就明白实现阻塞功能的方法了，就是不让线程在就绪队列中出现就行了，这样线程便没有机会运行，也就是实现了线程的阻塞，是不是觉得好简单？

阻塞是线程自己发出的动作，也就是线程自己阻塞自己，并不是被别人阻塞的，阻塞是线程主动的行为。已阻塞的线程是由别人来唤醒的，唤醒是被动的。如果您对此感到奇怪，我再多解释几句。

先说一下阻塞。

调度器只负责挑选“有运行意愿、准备好运行”的线程上处理器运行，如果线程没有运行的意愿，调度器也不能强迫它，这倒不是因为“民主”，其实线程巴不得总在处理器上运行，永远不下来，所以它不是不想运行，而是不能运行。原因是运行的条件不具备，停止运行实属无奈，因此，即使是强制运行也没有好下场。

调度器的功能只是去挑选哪个线程运行，即使再差的调度算法也会保证每个线程都有运行的机会，哪怕只是运行几个时钟周期。因此，调度器并不决定线程是否可以运行，只是决定了运行的时机，线程可否运行是由线程自己把控的。当线程被换上处理器运行后，在其时间片内，线程将主宰自己的命运。阻塞是一种意愿，表达的是线程运行中发生了一些事情，这些事情通常是由于缺乏了某些运行条件造成的，以至于线程不得不暂时停下来，必须等到运行的条件再次具备时才能上处理器继续运行。因此，阻塞发生的时间是在线程自己的运行过程中，是线程自己阻塞自己，并不是被谁阻塞。

说一下唤醒。

已被阻塞的线程是无法运行的，属于睡梦中，因此它只能祈祷有个“大恩人”来唤醒它，否则它永远没有运行的机会。这个“大恩人”便是锁的持有者，它释放了锁之后便去唤醒在它后面因获取该锁而阻塞的线程。因此唤醒已阻塞的线程是由别的线程，通常是锁的持有者来做的。

值得注意的是线程阻塞是线程执行时的“动作”，因此线程的时间片还没用完，在唤醒之后，线程会继续在剩余的时间片内运行，调度器并不会将该线程的时间片“充满”，也就是不会再用线程的优先级 priority 为时间片 ticks 赋值。因为阻塞是线程主动的意愿，它也是“迫于无奈”才“慷慨”地让出处理器资源给其他线程，所以调度器没必要为其“大方”而“赏赐”它完整的时间片。

初次接触信号量会觉得很抽象，因此咱们看看这两个函数的实现吧。请见代码 10-2。

代码 10-2 （project/c10/b/thread/thread.c）

```
128 /* 当前线程将自己阻塞，标志其状态为 stat. */
129 void thread_block(enum task_status stat) {
130 /* stat 取值为 TASK_BLOCKED、TASK_WAITING、TASK_HANGING，
也就是只有这三种状态才不会被调度*/
131    ASSERT(((stat == TASK_BLOCKED) || \
(stat == TASK_WAITING) || \
(stat == TASK_HANGING)));
132   enum intr_status old_status = intr_disable();
133    struct task_struct* cur_thread = running_thread();
134    cur_thread->status = stat; // 置其状态为 stat
135    schedule();                // 将当前线程换下处理器
136 /* 待当前线程被解除阻塞后才继续运行下面的 intr_set_status */
```

```
137    intr_set_status(old_status);
138 }
139 
140 /* 将线程 pthread 解除阻塞 */
141 void thread_unblock(struct task_struct* pthread) {
142    enum intr_status old_status = intr_disable();
143      ASSERT(((pthread->status == TASK_BLOCKED) || (pthread->status == TASK_WAITING) ||
(pthread->status == TASK_HANGING)));
144    if (pthread->status != TASK_READY) {
145       ASSERT(!elem_find(&thread_ready_list, &pthread->general_tag));
146       if (elem_find(&thread_ready_list, &pthread->general_tag)) {
147          PANIC("thread_unblock: blocked thread in ready_list\n");
148       }
149       list_push(&thread_ready_list, &pthread->general_tag);
                    // 放到队列的最前面，使其尽快得到调度
150       pthread->status = TASK_READY;
151    }
152    intr_set_status(old_status);
153 }
```

我们先看 thread_block，它接受一个参数 stat，stat 是线程的状态，它的取值为“不可运行态”，函数功能是将当前线程的状态置为 stat，从而实现了线程的阻塞。

强调一下，函数中不包括指向其他线程的指针，因此是当前线程自己调用 thread_block 阻塞自己，线程并不能被别人强制阻塞，在时间片内，任何线程被换下处理器都是出于“主动”和“自愿”。

stat 取值范围是 TASK_BLOCKED、TASK_WAITING 和 TASK_HANGING，这三个就是上面所说的“不可运行态”，后两个将来会用到。因此在第 131 行通过 ASSERT 做了限制。在咱们系统中只有 status 为 TASK_RUNNING 的线程才可以被添加到就绪队列 thread_ready_list，才有机会上处理器运行。

当前运行线程的 status 必然是 TASK_RUNNING，此状态的线程在调度器中会被重新加到就绪队列中。由于咱们要实现的功能是线程阻塞，也就是当前线程暂时不能运行。我们达到这一目的的原理是：让调度器 schedule 无法再调度它，也就是当前线程不能再被加到就绪队列 thread_ready_list 中。

回顾一下，在调度器 schedule 函数中，它会对当前线程的 status 判断。若当前线程的 status 为 TASK_RUNNING，这说明当前线程只是时间片到了，此次调度并不是由于阻塞而引发的，因此会将其重新加入到就绪队列中并置其状态为 TASK_READY。

为了不让其再被调度，必须将其 status 置为非 TASK_RUNNING，也就是函数 thread_block 的参数 stat。于是在第 133～134 行获取当前线程，并置其 status 为 stat。之后便调用 schedule 重新调度下一任务。

在调用 schedule 之后，下面的中断状态恢复代码 intr_set_status(old_status)本次便没机会执行了，只有在当前线程被唤醒后才会被执行到。

下面看看唤醒的函数 thread_block。

函数 thread_unblock 与 thread_block 的功能相反，它将某线程解除阻塞，也就是唤醒某线程。被阻塞的线程已无法运行，无法自己唤醒自己，必须被其他线程唤醒，因此参数 pthread 指向的是目前已经被阻塞，又希望被唤醒的线程。函数 thread_unblock 是由当前运行的线程调用的，由它实施唤醒动作，它是被阻塞线程 pthread 的“救世主”。

被阻塞的线程，其状态肯定不是 TASK_READY，为保险起见，我们还是判了一下，也就是第 144 行的 if (pthread->status != TASK_READY)仅仅是让我们更放心。

按常理说就绪队列中不会出现已阻塞的线程，但还是担心有些意外情况会导致阻塞的线程已经在就绪队列中，为防止已经在就绪队列中的线程再次被添加，在第 145 行通过 ASSERT 判断。

毕竟 ASSERT 只是调试期间用的，最后会把它去掉。但是这个判断我们还是需要的，因此第 146 行由 if 结合 PANIC 宏再次判断这种重复添加的情况。

接着通过 list_push 将阻塞的线程重新添加到就绪队列，这里用 list_push 是将线程添加到就绪队列的队首，因此保证这个睡了很久的线程能被优先调度。

最后再将线程的 status 置为 TASK_READY，至此，线程重新回到了就绪队列，它有再被调度的机会了，也就是实现了唤醒。

好，锁的基础部件已经说完了，本节至此结束，下节咱们开始实现锁。

10.1.5 锁的实现

不骗大伙儿，这回咱们真地要实现锁了，本节在 thread 目录下创建了 sync.h 和 sync.c 两个文件，我们的同步机制信号量和锁在这里面定义。现在说干货，请见代码 10-3。

代码 10-3 （project/c10/b/thread/sync.h）

```
 1 #ifndef __THREAD_SYNC_H
 2 #define __THREAD_SYNC_H
 3 #include "list.h"
 4 #include "stdint.h"
 5 #include "thread.h"
 6
 7 /* 信号量结构 */
 8 struct semaphore {
 9     uint8_t  value;
10     struct   list waiters;
11 };
12
13 /* 锁结构 */
14 struct lock {
15     struct   task_struct* holder;       // 锁的持有者
16     struct   semaphore semaphore;       // 用二元信号量实现锁
17     uint32_t holder_repeat_nr;          // 锁的持有者重复申请锁的次数
18 };
略…
```

代码 10-3 中定义了两个结构，其中 struct semaphore 为信号量结构，根据我们之前的描述，信号量要有初值，因此该结构中包含了成员 value，对信号量进行 down 操作时，若信号量值为 0 就会阻塞线程，因此该结构中还包括了成员 waiters，用它来记录在此信号量上等待（阻塞）的所有线程。再次重申一次，信号量仅是个编程理念，是个程序设计结构，只要具备信号量初值和等待线程这两个必要元素就可以，其实现形式无具体要求。

第二个结构是 struct lock，这是锁结构。谁成功申请了锁，就应该记录锁被谁持有，这是用成员 holder 记录的，表示锁的持有者。前面已介绍过，我们的锁是基于信号量来实现的，因此锁结构中必须包含一个信号量成员，这里就是 semaphore，它就是信号量结构体 struct semaphore 实例。将来此信号量的初值会被赋值为 1，也就是用二元信号量实现锁。成员 holder_repeat_nr 用来累积锁的持有者重复申请锁的次数，释放锁的时候会参考此变量的值。原因是一般情况下我们应该在进入临界区之前加锁，但有时候可能持有了某临界区的锁后，在未释放锁之前，有可能会再次调用重复申请此锁的函数，这样一来，内外层函数在释放锁时会对同一个锁释放两次，为了避免这种情况的发生，用此变量来累积重复申请的次数，释放锁时会根据变量 holder_repeat_nr 的值来执行具体动作。

头文件就介绍到此，下面看实现文件 sync.c。请见代码 10-4-1。

代码 10-4-1 （project/c10/b/thread/sync.c）

```
…略
 7 /* 初始化信号量 */
 8 void sema_init(struct semaphore* psema, uint8_t value) {
 9    psema->value = value;          // 为信号量赋初值
10    list_init(&psema->waiters);    // 初始化信号量的等待队列
11 }
12
13 /* 初始化锁 plock */
14 void lock_init(struct lock* plock) {
15    plock->holder = NULL;
16    plock->holder_repeat_nr = 0;
17    sema_init(&plock->semaphore, 1);  // 信号量初值为 1
18 }
19
20 /* 信号量 down 操作 */
21 void sema_down(struct semaphore* psema) {
22 /* 关中断来保证原子操作 */
23    enum intr_status old_status = intr_disable();
```

```
24    while(psema->value == 0) {   // 若 value 为 0，表示已经被别人持有
25        ASSERT(!elem_find(&psema->waiters, \
&running_thread()->general_tag));
26        /* 当前线程不应该已在信号量的 waiters 队列中 */
27        if (elem_find(&psema->waiters, &running_thread()->general_tag)) {
28            PANIC("sema_down: thread blocked has been in waiters_list\n");
29        }
30 /* 若信号量的值等于 0，则当前线程把自己加入该锁的等待队列，
   然后阻塞自己 */
31        list_append(&psema->waiters, &running_thread()->general_tag);
32        thread_block(TASK_BLOCKED);    // 阻塞线程，直到被唤醒
33    }
34 /* 若 value 为 1 或被唤醒后，会执行下面的代码，也就是获得了锁*/
35    psema->value--;
36    ASSERT(psema->value == 0);
37 /* 恢复之前的中断状态 */
38    intr_set_status(old_status);
39 }
```

咱们从上往下说，sema_init 函数接受两个参数，psema 是待初始化的信号量，value 是信号量的初值，函数功能是将信号量 psema 初值初始化为 value。锁是用信号量来实现的，因此锁的初始化中会调用 sema_init。

函数 lock_init 接受一个参数，plock 是待初始化的锁。函数功能是将锁的持有者 holder 置为空，将持有者重复申请次数累积变量 holder_repeat_nr 置为 0，并调用 sema_init(&plock->semaphore, 1)将锁使用的信号量初值赋值为 1，这样锁中的信号量就成为了二元信号量。

函数 sema_down 是核心函数，乃重中之重。它接受一个参数，psema 是待执行 down 操作的信号量。函数功能就是在信号量 psema 上执行个 down 操作。

为保证 down 操作的原子性，在函数开头便通过 intr_disable 关了中断。

当信号量的值为 1 时，down 操作才会成功返回，否则就在该信号上阻塞。这里通过 while(psema->value == 0)判断信号量是否为 0，如果为 0，就进入 while 的循环体做两件事。

1．将自己添加到该信号量的等待队列中。对应的代码为：

```
list_append(&psema->waiters, &running_thread()->general_tag);
```

2．将自己阻塞，状态为 TASK_BLOCKED。对应的代码为：

```
thread_block(TASK_BLOCKED);
```

按理说，既然当前线程已处于活动中，也就是状态为 TASK_RUNNING，当前线程就不会出现在此信号量的等待队列中，否则重复添加的话会破坏队列。因此，为避免异常情况发生，在做以上两件事之前，还是用 ASSERT 和 if 排除当前线程已在等待队列中的情况。

如果信号量不为 0，也就是为 1（在咱们的应用中 value 要么为 0，要么为 1），或者之前为 0，现在线程被唤醒后已经为 1 了，则将信号量减 1，即 psema->value--。此时 value 的值应该为 0，因此用 ASSERT (psema->value == 0)做判断。这是为防止程序出错时，出现 value 大于 1 的情况。

不知道大伙儿是否奇怪，为什么上面在判断信号量是否为 0 时，用的是 while，而不是 if，while(psema->value == 0)和 if(psema->value == 0)有什么不同吗？

锁本身也是公共的资源，大家也要通过竞争的方式去获得它，因此想要获得锁的线程不只一个，当阻塞的线程被唤醒后，也不一定就能获得资源，只是再次获得了去竞争锁的机会而已，所以判断信号量的值最好用 while，而不是用 if。直观上理解，就是判断的次数不同，线程用 while，可以在被唤醒后再次做条件判断，而 if 则只能判断一次，这就是最大的区别。似乎是说了等于没说，哈哈，我知道，待兄弟给您举个例子。

比如现在有 3 个线程，分别是 t_a，t_b，t_c。假如目前锁由线程 t_a 持有，因此锁中信号量的值为 0。t_b 也来申请锁，但由于信号量的值为 0，故 t_b 阻塞。当线程 t_a 执行完临界区代码后它会释放锁，释放锁的操作包括两件事。

（1）使信号量的值恢复为 1。

（2）唤醒阻塞的线程。

这里将会唤醒线程 t_b。此后，t_b 将会在将来某一时间恢复运行，继续抢锁。正巧的是线程 t_c 也来

申请锁了，它比 t_b 先得到调度，因此，它抢先获得了锁。此时信号量又变成了 0。时光飞逝，终于线程 t_b 又被调度上处理器了，注意，下面要见分晓了。

- 如果之前是用 if(psema->value == 0)来判断信号量的 value 是否为 0，线程 t_b 醒来的第一件事就是：

执行 psema->value--，使 value 减 1。但它不知 value 之前已经被 t_c 置为 0 了，并不是 1，您看，这就错了吧。虽然 value 是 uint8_t 类型，值不会为负，但值会变成 8 位宽度的最大值 255，这更不对了。

- 如果之前是用 while(psema->value == 0)判断信号量的 value 是否为 0，那线程 t_b 醒来的第一件事就是：

再次执行 while 循环的判断 psema->value == 0，确认下 value 是否变成 1 了，如果依然为 0，继续执行阻塞相关的工作。如果不为 0，则执行 psema->value--，使 value 减 1 为 0，表示获得了锁。

您看，锁的竞争者太多了，并不是说线程在唤醒后，锁就在那闲着等着它来拿。必须确保锁是闲着的才行，因此线程醒来后依然对信号量做判断，我们必须用 while。

不过话说回来了，咱们这里是可以用 if 代替的，只不过用 while 更通用。可以用 if 代替的原因是咱们在后面您就会看到，信号量的 up 操作是通过 thread_unblock 唤醒线程的，而 thread_unblock 会把阻塞的线程放在就绪队列的队首，因此会紧随当前锁的持有者之后调度，也就是当前锁持有者释放锁后，它会第一个获得锁。

下面继续看代码 10-4-2。

代码 10-4-2 （project/c10/b/thread/sync.c）

```
41 /* 信号量的 up 操作 */
42 void sema_up(struct semaphore* psema) {
43 /* 关中断，保证原子操作 */
44    enum intr_status old_status = intr_disable();
45    ASSERT(psema->value == 0);
46    if (!list_empty(&psema->waiters)) {
47        struct task_struct* thread_blocked = elem2entry(struct task_struct, \
          general_tag, list_pop(&psema->waiters));
48        thread_unblock(thread_blocked);
49    }
50    psema->value++;
51    ASSERT(psema->value == 1);
52 /* 恢复之前的中断状态 */
53    intr_set_status(old_status);
54 }
55
56 /* 获取锁 plock */
57 void lock_acquire(struct lock* plock) {
58 /* 排除曾经自己已经持有锁但还未将其释放的情况 */
59    if (plock->holder != running_thread()) {
60        sema_down(&plock->semaphore);   // 对信号量 P 操作，原子操作
61        plock->holder = running_thread();
62        ASSERT(plock->holder_repeat_nr == 0);
63        plock->holder_repeat_nr = 1;
64    } else {
65        plock->holder_repeat_nr++;
66    }
67 }
68
69 /* 释放锁 plock */
70 void lock_release(struct lock* plock) {
71    ASSERT(plock->holder == running_thread());
72    if (plock->holder_repeat_nr > 1) {
73        plock->holder_repeat_nr--;
74        return;
75    }
76    ASSERT(plock->holder_repeat_nr == 1);
77
78    plock->holder = NULL;        // 把锁的持有者置空放在 V 操作之前
79    plock->holder_repeat_nr = 0;
80    sema_up(&plock->semaphore);  // 信号量的 V 操作，也是原子操作
81 }
```

函数 sema_up 接受一个参数，psema 是待执行 up 操作的信号量。函数功能是将信号量的值加 1。

函数内部的操作也要保证原子性，因此在函数的开头也执行了 intr_disable 函数关中断。

sema_up 是使信号量加 1，这表示有信号资源可用了，也就是其他线程可以申请锁了，因此在信号量的等待队列 psema->waiters 中通过 list_pop 弹出队首的第一个线程，并通过宏 elem2entry 将其转换成 PCB，存储到 thread_blocked 中。然后通过 thread_unblock(thread_blocked)将此线程唤醒。

在将线程唤醒后，接下来将信号量值加 1，即代码 psema->value++。

提醒一下，所谓的唤醒并不是指马上就运行，而是重新加入到就绪队列，将来可以参与调度，运行是将来的事。而且当前是在关中断的情况下，所以调度器并不会被触发。因此不用担心线程已经加到就绪队列中，但 value 的值还没变成 1 会导致出错。

最后通过 intr_set_status(old_status)恢复之前的中断状态。

函数 lock_acquire 接受一个参数，plock 是所要获得的锁，函数功能是获取锁 plock。有时候，线程可能会嵌套申请同一把锁，这种情况下再申请锁，就会形成死锁，即自己在等待自己释放锁。因此，在函数开头先判断自己是否已经是该锁的持有者，即代码 if (plock->holder != running_thread())。如果持有者已经是自己，就将变量 holder_repeat_nr++，除此之外什么都不做，然后函数返回。如果自己尚未持有此锁的话，通过 sema_down(&plock->semaphore)将锁的信号量减 1，当然在 sema_down 中有可能会阻塞，不过早晚会成功返回的。成功后将当前线程记为锁的持有者，即 plock->holder = running_thread()，然后将 holder_repeat_nr 置为 1，表示第 1 次申请了该锁。

函数 lock_release 只接受一个参数，plock 指向待释放的锁，函数功能是释放锁 plock。当前线程应该是锁的持有者，所以用 ASSERT 判断了一下。如果持有者的变量 holder_repeat_nr 大于 1，这说明自己多次申请该锁，此时还不能真正将锁释放，因此只是将 holder_repeat_nr--，随后返回。如果锁持有者的变量 holder_repeat_nr 为 1，说明现在可以释放锁了，通过代码 plock->holder = NULL 将持有者置空，随后将 holder_repeat_nr 置为 0，最后通过“sema_up(&plock->semaphore)”将信号量加 1，自此，锁被真正释放。

注意，要把持有者置空语句“plock->holder = NULL”放在 sema_up 操作之前。原因是释放锁的操作并不在关中断下进行，有可能会被调度器换下处理器。若 sema_up 操作在前的话，sema_up 会先把 value 置 1，若老线程刚执行完 sema_up，还未执行“plock->holder = NULL”便被换下处理器，新调度上来的进程有可能也申请了这个锁，value 为 1，因此申请成功，锁的持有者 plock->holder 将变成这个新进程的 PCB。假如这个新线程还未释放锁又被换下了处理器，老线程又被调度上来执行，它会继续执行“plock->holder = NULL”，将持有者置空，这就乱了。

好啦，有关锁的内容就介绍完了，咱们下一节得想办法找个应用的环境，还是拿打印字符串测试吧，看看咱们的锁能否成功解决竞争条件。

10.2 用锁实现终端输出

本节我们实现输出。

大家若是用过 Linux，肯定都熟悉 Linux 终端，它是咱们与 Linux 系统交互的界面。

终端也称为控制台，这是在计算机历史中遗留下来的概念。在过去计算机还是奢侈品，为了充分利用计算机资源，允许多个用户同时连接到机器上，这类似 Windows 多用户的概念，为的是让更多的用户能够控制计算机，因此终端便称为控制台。

为了不让大伙儿失望，提前说一下，这里不打算实现这种类似 Linux 中的终端，但还是要简要介绍下终端的实现原理。

终端的构造原理很简单，就是把用户键入的命令传送到主机，待主机运算完成后再将结果送回给用户，终端不提供任何额外功能，仅是个显示窗口。

按理说一个用户就该配有一个显示器，这样才是多用户该有的“体验”，但那个时代太穷了，计算机对那时的人们来说就如同现代人们想拥有私人飞机一样奢侈，因此那时候的人们不可能会那么“任性”，为了让更多的人同时使用计算机，必须在同一个显示器下实现多用户，也就是分别为每个用户虚拟出一个“显示器”，这就是虚拟终端的由来，因此每个控制台其实就是个虚拟终端，用户看到的屏幕是由软件虚拟

出来的。

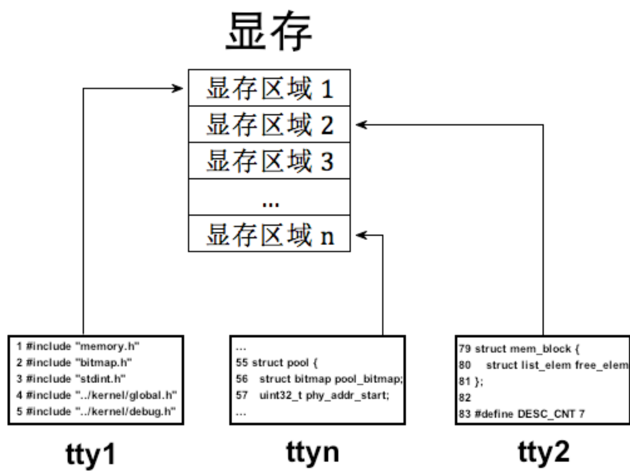
虚拟终端就是我们熟知的 tty，据说 tty 原指电传打字机，即 TeleTYpes，它是一种用打字机键盘通过串行线发送和接收信息的设备，后来被键盘和显示器取代了，因此称为 tty 翻译为终端更合适。我们登录系统后，就会在后台运行一个 tty 进程，如图 10-7 所示。

```
[root@localhost ~]# ps
  PID TTY          TIME CMD
 2014 tty1     00:00:00 bash
 2033 tty1     00:00:00 ps
[root@localhost ~]#
```

▲图 10-7　tty

这种虚拟终端是如何实现的呢？还是要依赖于硬件本身的功能。

我们知道屏幕在不同模式下显示的字符数是有限的，比如咱们所用的 80*25 模式只会显示 2000 个字，因此屏幕不能一次性把显存中的全部数据显示出来，为此，显卡提供了这两个寄存器“Start Address High Register”和“Start Address Low Register”来设置数据在显存中的起始地址。起始地址是用 16 位来表示的，它们分别设置显存地址的 15～8 位和 7～0 位。因此，我们可以把不同的 16 位地址分别写入这两个寄存器，从而实现将显存分块显示的目的，也就是实现了虚拟终端。由此可见，虽然多个虚拟终端共用同一个显示器，也就是共享同一片显存，但用户之间能够互不干扰，就是因为每个虚拟终端显示的是显存中的不同区域，如图 10-8 所示。



▲图 10-8　tty 分属显存中不同区域

原理就是这样，它们如何使用呢？

Linux 启动后一般会有 7 个界面可选，1 个是图形界面，也称为 Xwindows，我们可以像操作 Windows 那样用鼠标点击图形控件与系统交互。还有 6 个纯文本的控制台界面，我们的 shell 便运行在控制台界面中，因此我们才可以输入命令与 Linux 内核交互。一般按下 Alt+F1、Alt+F2 等组合键则会切换到不同的控制台（各 Linux 发行版本会有差异），个人感觉上就像切换用户一样，每次切换，后台对应的操作就是变换那两个寄存器中的起始地址。

现在咱们操作 Linux，都是通过 ssh 远程连上去，除了去机房外，很少有直接在机器上登录系统的。包括我自己装虚拟机的时候，都是另装个 ssh 工具连接到虚拟机，习惯了 ssh 客户端的便利。顺便说一下，这种从远程连接到 Linux 主机的终端称为 pts，如图 10-9 所示。

```
[root@localhost ~]# who
root     tty1         2015-01-16 13:52
work     pts/0        2015-01-16 14:14 (192.168.56.1)
[root@localhost ~]#
```

▲图 10-9　pts 和 tty

who 命令可以显示当前登录到系统中的用户，默认情况下显示用户名、登录方式（本地 tty1，还是远程 pts）及登录时间等。

结果中第 1 行的“root tty/1”是在本地用 root 登录的，就是上面所说的图 10-7。第 2 行的“work pts/0”是用 work 账号在远程登录的，后面有显示远程 IP：192.168.56.1。

我们不需要多个终端，因此并不打算实现这种“真正”的“虚拟”终端（这么说似乎有点矛盾），目前只要 1 个终端便能满足我们的需求。那我们本节要干吗呢？

在我们的系统中没有复杂的显卡寄存器操作，我们只有一个终端，因此所有的输出都往这一个屏幕上挤，这就容易让输出凌乱不堪。为了让这一个屏幕上的内容井然有序，既然我们已经实现了锁，我们可以通过锁实现输出互斥，这样屏幕上的字符就会井然有序，终于锁派上用场了。

虽然我们没有真正的多控制台，但是我们把终端当成设备来对待，终端就是我们的标准输出设备，因此我们本节要构造出一个终端设备，以后的打印输出就通过它了。

好啦，不再卖关子了，上菜，我们在 device 目录中创建了文件 console.c，请见代码 10-5。

代码 10-5　（project/c10/b/device/console.c）

```
1 #include "console.h"
2 #include "print.h"
```

```
 3 #include "stdint.h"
 4 #include "sync.h"
 5 #include "thread.h"
 6 static struct lock console_lock;    // 控制台锁
 7
 8 /* 初始化终端 */
 9 void console_init() {
10    lock_init(&console_lock);
11 }
12
13 /* 获取终端 */
14 void console_acquire() {
15    lock_acquire(&console_lock);
16 }
17
18 /* 释放终端 */
19 void console_release() {
20    lock_release(&console_lock);
21 }
22
23 /* 终端中输出字符串 */
24 void console_put_str(char* str) {
25    console_acquire();
26    put_str(str);
27    console_release();
28 }
29
30 /* 终端中输出字符 */
31 void console_put_char(uint8_t char_asci) {
32    console_acquire();
33    put_char(char_asci);
34    console_release();
35 }
36
37 /* 终端中输出十六进制整数 */
38 void console_put_int(uint32_t num) {
39    console_acquire();
40    put_int(num);
41    console_release();
42 }
```

文件 console.c 还是比较简单的，简洁到我都觉得没什么可说的，哈哈，毕竟它不是真正意义上的终端，甚至连伪终端都算不上，我们只是通过它让输出变得更整洁。您看到了，它就是对各种锁操作的封装，完全就是锁的应用。

文件中定义的 console_lock 是终端锁，对终端的所有操作都是围绕申请这个锁展开的。它必须是全局唯一的，因此类型是静态 static。

文件开头定义的初始化终端函数 console_init、获取终端函数 console_acquire 和释放终端函数 console_release 较简单，一看就明白。后面定义了三个输出函数，console_put_str、console_put_char 和 console_put_int，它们分别是 put_str、put_char 和 put_int 的封装，用于在终端中打印字符串、单个字符和十六进制整数。这三个终端打印函数共性是：在输出前后增加了 console_acquire 去获取终端，console_release 去释放终端，以此来实现互斥。

说完了，看看如何应用吧，先将 console_init 添加到 init_all 中，如代码 10-6 所示。

代码 10-6　（project/c10/b/kernel/init.c）

```
 9 /*负责初始化所有模块 */
10 void init_all() {
11    put_str("init_all\n");
12    idt_init();    // 初始化中断
13    mem_init();    // 初始化内存管理系统
14    thread_init();  // 初始化线程相关结构
15    timer_init();  // 初始化 PIT
16    console_init(); //控制台初始化最好放在开中断之前
17 }
```

除了在 init_all 中添加了 console_init 外，其他模块的初始化顺序也有所调整，只是这样的顺序看着顺眼。

下面是应用测试，咱们把 main.c 中的字符串输出函数 put_str 统统改为通过终端函数 console_ put_str

输出，如代码 10-7 所示。

代码 10-7 （project/c10/b/kernel/main.c）

```
…略
10 int main(void) {
11      put_str("I am kernel\n");
12      init_all();
13
14      thread_start("k_thread_a", 31, k_thread_a, "argA ");
15      thread_start("k_thread_b", 8, k_thread_b, "argB ");
16
17      intr_enable();
18      while(1) {
19         console_put_str("Main ");
20      };
21      return 0;
22 }
23
24 /* 在线程中运行的函数 */
25 void k_thread_a(void* arg) {
26 /* 用 void*来通用表示参数，
   被调用的函数知道自己需要什么类型的参数，自己转换再用 */
27      char* para = arg;
28      while(1) {
29         console_put_str(para);
30      }
31 }
32
33 /* 在线程中运行的函数 */
34 void k_thread_b(void* arg) {
35 /* 用 void*来通用表示参数，
   被调用的函数知道自己需要什么类型的参数，自己转换再用 */
36      char* para = arg;
37      while(1) {
38         console_put_str(para);
39      }
40 }
```

编译，运行结果如图 10-10 所示。

▲图 10-10 通过终端输出字符串

您看，字符串输出的顺序是：

“argA”->“Main”->“argB”->“Main”->“argA”->“main”，

字符串打印顺序不再随线程调度的顺序一致，而且每组字符串的数量较之前关中断的版本多了一倍以上，因此线程阻塞起作用了。

好啦，我们的终端到这就算完成了，已经能够满足咱们需求，今后也不再更新它了。

兄弟们辛苦了，咱们下节见。

10.3 从键盘获取输入

上节中的终端虽然简陋，但也算是个“输出”了，本节咱们再实现个“输入”，即从键盘获取键入的字符。

时至今日，我们 80 后这一代人用过的键盘有三种类型：PS/2 键盘、USB 键盘和蓝牙键盘。虽然 PS/2 键盘已经很少有人用了，但后来的新型键盘都是基于它发展起来的，基础原理是不变的，因为要考虑兼容性，所以肯定不能将过去的成绩全盘推翻而重新发明一套做法，这就像 Intel 处理器虽然早已经发展到 I7 好多年，但大学教材中依然用 8086 处理器作为学习汇编指令的模型一样，原理不变，经典，经得起考验。因此，咱们也以经典的 PS/2 键盘为例展开介绍。

10.3.1　键盘输入原理简介

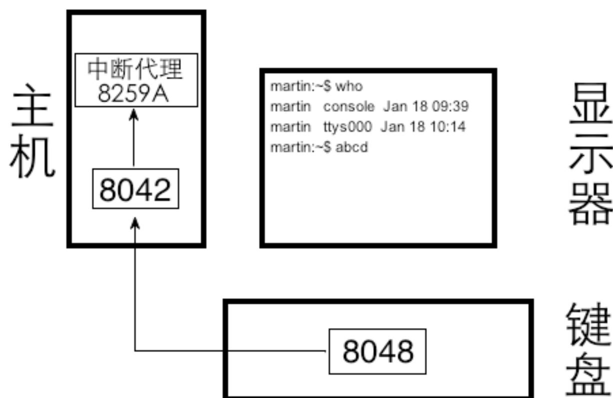
计算机是个系统，系统是指由各功能独立的模块组成的整体，相当于在内部按功能分层，一个模块就像个功能独立的黑盒子，上下游模块之间可依赖，相互提供数据。在所有模块的配合下，使这个系统作为整体对外提供服务。

因此，我们平时所熟悉的键盘操作，也是由独立的模块分层实现的，但是并不是简单地由键盘把数据塞到主机里，这涉及两个功能独立的芯片的配合。

键盘是个独立的设备，在它内部有个叫作键盘编码器的芯片，通常是 Intel 8048 或兼容芯片，它的作用是：每当键盘上发生按键操作，它就向键盘控制器报告哪个键被按下，按键是否弹起。

这个键盘控制器可并不在键盘内部，它在主机内部的主板上，通常是 Intel 8042 或兼容芯片，它的作用是接收来自键盘编码器的按键信息，将其解码后保存，然后向中断代理发中断，之后处理器执行相应的中断处理程序读入 8042 处理保存过的按键信息。

它们的关系如图 10-11 所示。



▲图 10-11　8048-8042-8259A 的关系

8048 是键盘上的芯片，其主要责任就是监控哪个键被按下。当键盘上发生按键操作时，8048 当然知道是哪个键被按下。但光它自己知道还不行，它毕竟要将按键信息传给 8042，必须得让 8042 知道到底是按下了哪个键，为此 8048 必然要和 8042 达成一个协议，这个协议规定了键盘上的每个物理键对应的唯一数值，说白了就是对键盘上所有的按键进行编码，为每个按键分配唯一的数字，这样双方都知道了每个数值代表哪个键。当某个键被按下时，8048 把这个键对应的数值发送给 8042，8042 根据这个数值便知道是哪个键被按下了。

您想，键盘上那么多的键，每个键都要有数值，因此所有按键对应的数值便组成了一张“按键-数值”编码映射表，当然人家可不是这么俗套的名字，这张表的官方名称为键盘扫描码。

回想一下，当我们想在屏幕上连续输入多个相同的字符时，我们通常都是按下某个按键不松手，结束输入时才松手，也就是键被弹起。比如在聊天工具里输入一堆“………”表示无语的时候，或者刷新网页时长按 F5 键时，总之，很少有人一下一下地按键。当我们松开手，按键被弹起时就表示输入完成，也就是说，我们也得让 8042 知道何时按键被弹起，也就是击键操作何时结束，这样 8042 才知道用户在一次持续按键操作中到底输入了多少个相同的字符。因此，键盘扫描码中不仅仅要记录按键被按下时对应的编码，也要记录按键被松开（弹起）时的编码。总之在输入框中看似随意的打字行为，在幕后都有一些硬件在一丝不苟地完成繁重的工作。

一个键的状态要么是按下，要么是弹起，因此一个键便有两个编码，按键被按下时的编码叫通码，也

就是表示按键上的触点接通了内部电路，使硬件产生了一个码，故通码也称为makecode。按键在被按住不松手时会持续产生相同的码，直到按键被松开时才终止，因此按键被松开弹起时产生的编码叫断码，也就是电路被断开了，不再持续产生码了，故断码也称为breakcode。一个键的扫描码是由通码和断码组成的。

无论是按下键，或是松开键，当键的状态改变后，键盘中的8048芯片把按键对应的扫描码（通码或断码）发送到主板上的8042芯片，由8042处理后保存在自己的寄存器中，然后向8259A发送中断信号，这样处理器便去执行键盘中断处理程序，将8042处理过的扫描码从它的寄存器中读取出来，继续进行下一步处理。

这个键盘中断处理程序是咱们程序员负责编写的，值得注意的是我们只能得到键的扫描码，并不会得到键的ASCII码，扫描码是硬件提供的编码集，ASCII是软件中约定的编码集，这两个是不同的编码方案。我们的键盘中断处理程序是同硬件打交道的，因此只能得到硬件提供的扫描码，但我们可以将得到的“硬件”扫描码转换成对应的“软件”ASCII码。假如我们在键盘上按下了空格键，我们在键盘中断处理程序中只能得到空格键的扫描码，该扫描码是0x39（后面会有详细介绍），而不是空格键的ASCII码0x20。

按键的表现行为是字符处理程序负责的，键盘的中断处理程序便充当了字符处理程序。一般的字符处理程序使用字符编码来处理字符，比如ASCII码，因此我们可以在中断处理程序中将空格的扫描码0x39转换成ASCII码0x20，然后将ASCII码0x20交给我们的put_char函数，将ASCII码写入显存，也就是输出到屏幕。因此，按下空格键可以在屏幕上输出一个空格，就是这么来的。

但咱们也可以不走寻常路，完全可以将扫描码转换成任意字符的ASCII码，只是这样不符合我们的习惯。比如将空格键的扫描码处理成字符g的ASCII码，也就是按空格键时相当于键入字符g，这显然不合理。

因此，按键产生什么样的行为，完全是由字处理软件负责的，我们也按照约定俗成的规则，按下什么键就产生对应的字符的ASCII码。

以上仅是个大致的原理，主机上的8042芯片是如何处理来自键盘中8048芯片的扫描码的呢？这涉及到键盘扫描码的分类，咱们下节再说。

10.3.2 键盘扫描码

任何事物的发展都是新老交替的过程，发展过程中除了要吸取前辈的精华，摒弃它们的糟粕之外，还要考虑到兼容性，计算机能有今天的繁荣，可想而知，工程师们在兼容性方面付出了多大的努力。

键的扫描码是由键盘中的键盘编码器决定的，不同的编码方案便是不同的键盘扫描码，也就是说，相同的键在不同的编码方案下产生的通码和断码也是不同的（即使有相同的例外，也是巧合），不同的编码方案有不同的编码规则。

根据不同的编码方案，键盘扫描码有三套，分别称为scan code set 1、scan code set 2、scan code set 3。至于为什么有这么多套扫描码，一个可能的原因是：键盘扫描码就是对键盘中所有键的编码，编码的目的是为了方便工作，编码时要考虑后续的软硬件对编码处理时也要方便才行，因此编码是有规律的。早期键盘上的键不多，因此扫描码规模也小，旧扫描码的编码方法足够应对。后来计算机逐渐发展，尤其是对多媒体的支持越来越强大，键盘上支持的功能键就多了，再加上为了方便人们使用键盘，在键盘的另一侧也增加了功能相同的按键，比如左右都有 shift、alt、ctrl，于是原有的编码方法使得扫描码的处理变得不再高效，因此要跟着扩充、更新或者重新来一套新的，于是便有了新的键盘扫描码。

第一套键盘扫描码必然是由最早的键盘使用的，它就是XT键盘所用的扫描码。XT键盘如图10-12所示。

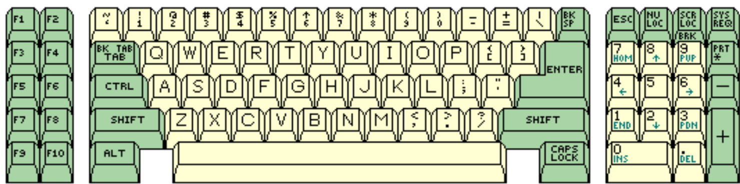


▲图10-12 IBM Personal Computer XT键盘

XT 键盘和今天的键盘确实长得很不一样，F1～F10这几个功能键都是在键盘的左边。另外，这张图

是黑白的，似乎也显得更加久远。

很多用户不喜欢 XT 键盘上的回车键和左 shift 键的位置，因此在 AT 键盘上有了改进，如图 10-13 所示。



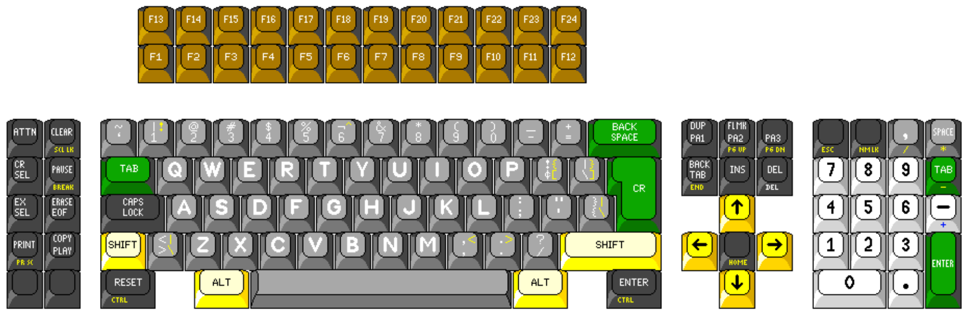
▲图 10-13　IBM Personal Computer AT 键盘

这张图是彩色的，显得很“近代”。

尽管 AT 键盘上的 backspace 键变得更小了，按起来也困难了，但键的位置布局重新，使计算机用户感到非常顺手，因此 AT 键盘一经推出后特别受计算机用户的欢迎。

AT 键盘上所用的扫描码就是第二套键盘扫描码，也就是现在键盘上普通使用的扫描码。您不要觉得键 F1～F10 的位置很怪异，这和扫描码无关，言外之意是不管键在哪里，键对应的扫描码是由 8048 编码的，键的扫描码和键的物理位置无关。

第三套键盘扫描码用在 IBM PS/2 系列高端计算机所用的键盘上，还有一些运行商业版 UNIX 系统的计算机也有用到它，不过这种键盘如今很少看到了，因此第三套键盘扫描码也很少到了。这种键盘如图 10-14 所示。



▲图 10-14　使用第三套键盘扫描码的键盘

综上所述，第二套键盘扫描码几乎是目前所使用的键盘的标准，因此大多数键盘向 8042 发送的扫描码都是第二套扫描码。

您看，我说的是现在“大多数”键盘用的都是第二套扫描码，但也难免还有用第一套和第三套扫描码的键盘，相同按键在不同键盘扫描码中对应的编码（扫描码）不同，我们在中断处理程序中也得根据扫描码来判断按的是哪个键，那我们如何知道键盘用的是哪套键盘扫描码呢？

任何不兼容的两种事物都可以通过加一个“中间层”的方式解决兼容，这就是 8042 存在的理由之一，8042 是 8048 和 CPU 之间的中间层。

我们（程序员）不是不知道键盘用的是哪种扫描码吗，那好，只要 8042 知道就行。为了兼容第一套键盘扫描码对应的中断处理程序，不管键盘用的是何种键盘扫描码，当键盘将扫描码发送到 8042 后，都由 8042 转换成第一套扫描码，这就是我们上一节中所说的 8042 的“处理”。

因此，我们在键盘的中断处理程序中只处理第一套键盘扫描码就可以了。那我们介绍下第一套键盘扫描码。请见表 10-1。

表 10-1　　第一套键盘扫描码

键	通码	断码	键	通码	断码
以下是主键盘区及功能区					
<esc>	01	81	<caps lock>	3a	ba
F1	3b	bb	a	1e	9e

续表

键	通码	断码	键	通码	断码
以下是主键盘区及功能区					
F2	3c	bc	s	1f	9f
F3	3d	bd	d	20	a0
F4	3e	be	f	21	a1
F5	3f	bf	g	22	a2
F6	40	c0	h	23	a3
F7	41	c1	j	24	a4
F8	42	c2	k	25	a5
F9	43	c3	l	26	a6
F10	44	c4	:;	27	a7
F11	57	d7	"'	28	a8
F12	58	d8	<enter>	1c	9c
~·	29	a9	<L-Shift>	2a	aa
!1	02	82	z	2c	ac
@2	03	83	x	2d	ad
#3	04	84	c	2e	ae
$4	05	85	v	2f	af
%5	06	86	b	30	b0
^6	07	87	n	31	b1
&7	08	88	m	32	b2
*8	09	89	<,	33	b3
(9	0a	8a	>.	34	b4
)0	0b	8b	?/	35	b5
_-	0c	8c	<R-shift>	36	b6
+=	0d	8d	<L-ctrl>	1d	9d
<backspace>	0e	8e	<L-alt>	38	b8
<tab>	0f	8f	<space>	39	b9
q	10	90	<R-alt>	e0,38	e0,b8
w	11	91	<R-ctrl>	e0,1d	e0,9d
e	12	12			
r	13	93			
t	14	94			
y	15	95			
u	16	96			
i	17	97			
o	18	98			
p	19	99			
{[	1a	9a			
}]	1b	9b			
|\	2b	ab			
以下是附加键及小键盘区					
PrintScreen SysRq	e0,2a,e0,37	e0,b7,e0,aa	NumLock	45	c5
Scroll Lock	46	c6	/	e0,35	e0,b5
Pause Break	e1 1d 45 e1 9d c5	无	*	37	b7
Insert	e0,52	e0,d2	-	4a	ca

续表

键	通码	断码	键	通码	断码
以下是附加键及小键盘区					
Home	e0,47	e0,c7	7Home	47	c7
Page Up	e0,49	e0,c9	8Up	48	c8
Delete	e0,53	e0,d3	9PgUp	49	c9
End	e0,4f	e0,cf	4Left	4b	cb
Page Down	e0,51	e0,d1	5	4c	cc
←	e0,46	e0,c6	6Right	4d	cd
→	e0,4d	e0,cd	1End	4f	cf
↑	e0,48	e0,c8	2Down	50	d0
↓	e0,50	e0,d0	3PgDn	51	d1
			0Ins	52	d2
			.Del	53	d3
			+	4e	ce
			Enter	e0,1c	e0,9c

表有点长，表中的键是以它们在键盘上实际的位置顺序列出的：从左到右、从上到下。

在主键盘区中灰色的部分是为了区分键盘中单独的一行，另外的灰色部分是为了区分附加键和小键盘区。

任何键盘扫描码都是有规律的，这主要是为了处理方便。

大多数情况下第一套扫描码中的通码和断码都是 1 字节大小。您看，表 10-1 中的通码和断码，它们的关系是：断码 ＝0x80＋ 通码。顺便说一句，在第二套键盘扫描码中，一般的通码是 1 字节大小，断码是在通码前再加 1 字节的 0xF0，共 2 字节，我们的 8042 工作之一就是根据第二套扫描码中通码和断码的关系将它们解码，然后按照第一套扫描码中通码和断码的关系转换成第一套扫描码。

我们继续回来说第一套键盘扫描码。

对于通码和断码可以这样理解，它们都是一字节大小，最高位也就是第 7 位的值决定按键的状态，最高位若值为 0，表示按键处于按下的状态，否则为 1 的话，表示按键弹起。比如按键<Esc>，不管键盘向 8042 发送的是第几套扫描码，当我们按下它的时候，最终被 8042 转换成 0x1，当我们松开它的时候，最终会被 8042 转换成 0x80+0x1=0x81。

完整的击键操作包括两个过程，先是被按下，也许是被按下一瞬间，也许是持续保持被按下，然后是被松开，总之，按下的动作是先于松开发生的，因此每次按键时会先产生通码，再产生断码。比如我们按下字符 a 时，按照第一套键盘扫描码来说，先是产生通码 0x1e，后是产生断码 0x9e。

大家一定注意到了，有些按键的通码和断码都以 0xe0 开头，它们占 2 字节，甚至 Pause 键以 0xe1 开头，占 6 字节。原因是这样的，并不是一种键盘就要用一套键盘扫描码，最初第一套键盘扫描码是由 XT 键盘所使用的，它后来也被一些更新的键盘所使用。XT 键盘上的键很少，比如右边回车键附近就没有 alt 和 ctrl 键，这是在后来的键盘中才加进去的，因此表示扩展 extend，所以在扫描码前面加了 0xe0 作为前缀。比如在 XT 键盘上，左边有 alt 键，其通码为 0x38，断码为 0xb8。右边的 alt 键是后来在新的键盘上加进去的，因此，一方面为了表示都是同样功能的 alt 键，另一方面表示不是左边那个 alt，而是右边的 alt，于是这个扩展的 alt 键的扫描码便为“0xe0 和原来左边 alt 的扫描码”。因此，右边 alt 键的通码便为“0xe0,0x38”，断码为“0xe0,0xb8”。

松开是按下的逆过程，为了体现这个“逆”字，咱们看下截屏键 PrintScreen 的通码和断码。

PrintScreen 的通码是 4 字节，其值为：e0，2a，e0，37，给人的感觉是由两个扩展通码组成的，即“e0，2a”为一对，“e0，37”为一对。断码依然是 0x80+通码，但由于松开是按下的逆过程，故断码值和通码值的顺序相反，即 e0，b7，e0，aa。断码中的“e0，b7”对应通码中的“e0，37”，断码中的“e0，aa”对应通码中的“e0，2a”。也就是说，当我们按下截屏键 PrintScreen 时（假设键盘使用的是第一套键盘扫描码），键盘发送的通码是以 e0->2a->e0->37 的顺序，松开 PrintScreen 时，键盘发送的断码是以 e0->b7->e0->aa 的顺序。

虽然 8048 和 8042 都知道击键操作何时发生和结束，但我们程序员是如何知道的呢？最终的键盘操作是由咱们来处理的，我们必须知道击键何时发生何时结束，也就是得清楚击键的过程，这样咱们才知道用户到底敲了哪些按键。

为了让我们获取击键的过程，在每一次击键动作的“按下”、“按下保持”和“弹起”三个阶段，确切地说是每次 8048 向 8042 发扫描码的时候，8042 都会向中断代理（咱们是 8259A）发一次中断，即“键被按下”时发中断，“持续按着不松手”时会持续发中断，“松开手，键被弹起”时也发中断，因此，我们的键盘中断处理程序每次都会随着键盘发出的扫描码而去执行，也就是也会收到完整的击键过程，包括键的持续按压状态。

举个例子，通常在 Windows 下 ctrl+a 键是全选，这个按键过程是怎样的呢？也许您没注意过，但步骤基本上是这样的。

（1）ctrl 键先被按下。

（2）保持 ctrl 键按住不松手。

（3）按下 a 键。

（4）松开 ctrl 键。

（5）松开 a 键。

一般情况下，尽管我们是按照以上五个步骤完成的文本全选（也许有极少数同学的步骤 4 和步骤 5 要调换），但其实只要前三个步骤发生时，Windows 就会把当前文本区中的文本全部选中，即步骤四和步骤五并不重要，也就是无论这两个键哪个先被弹起，都不影响当前文本被选中，这是为何呢？

下面咱们分析一下细节，假设键盘使用的是第二套键盘扫描码。

步骤（1）中，假设我们按下的是左边的 ctrl 键，此时 8048 向 8042 发出了<L-ctrl>键的通码 0x14（这是第二套扫描码，可见图 10-15），8042 收到 0x14 后将其转换为第一套键盘扫描码，即 0x1d，将其保存到自己的输出缓冲区寄存器，然后 8042 向中断代理发中断，随后处理器执行键盘中断处理程序，键盘中断处理程序从 8042 的输出缓冲区寄存器中获取扫描码，即 0x1d。键盘处理程序会判断这次按下的是哪个键，它一看是<L-ctrl>的通码（其实是<R-ctrl>也无所谓，一般情况下左右 ctrl 都被认为是 ctrl 键。它们只是位置不同，并不代表是两个不同的功能，键被放在不同的位置是为方便人们操作，当然这只是传统的作法），它便在某个全局变量中记录 ctrl 键已被按下。

步骤（2）中，<L-ctrl>键持续按住不松手，因此 8048 会持续向 8042 发送 0x14，8042 每次都将其转换成第一套键盘扫描码 0x1d 并向中断代理发中断，每次键盘中断处理程序都会从 8042 中得到 0x1d。和步骤 1 中一样，键盘处理程序一看是<L-ctrl>的通码，依然会在全局变量中记录下 ctrl 键被按下，尽管之前已经按下了 ctrl，但不重要，键盘处理程序也许只记录上一次按下的是哪个，不关注之前按下了多少次同样的键，当然这取决于具体实现。

步骤（3）中，a 键被按下，此时 8048 会向 8042 发出 a 键的第二套键盘扫描码 0x1c（通码），8042 将其转换成第一套键盘扫描码 0x1e 后保存到自己的输出缓冲区寄存器，之后向中断代理发中断，键盘中断处理程序开始执行，从 8042 的输出缓冲区寄存器中获取到 0x1e。键盘处理程序判断这次按下的是 a 键，查看之前 ctrl 键已经被按下了（那个全局变量有记录），因此判断用户按下的是“ctrl+a”组合键。ctrl、alt、shift 等控制键一般是与下一次按键组合，这是由于微软给咱们培训的操作习惯，即控制键先被按下，其他普通键后被按下。这次按下的不是控制键，因此把记录 ctrl 键是否按下的全局变量清空。然后把“ctrl+a”这一消息上报给上层模块，上层模块判断这是要执行全选的功能，于是文本被全部选中。

步骤（4）中，<L-ctrl>键被松开，8048 向 8042 发送它的第二套键盘扫描码 0xf0 和 0x14（断码），前面有提起过，第二套键盘扫描码的断码一般是 2 字节，由固定的前缀 0xf0 和其通码组成。8042 将这两个字节转换成第一套键盘扫描码 0x9d（断码），随后发中断，键盘中断处理程序一看最高位为 1，这是断码，表示键被松开了，不管松开的是什么键，忽略，不做任何处理。

步骤（5）中，a 键被松开，8048 向 8042 发送它的第二套键盘扫描码 0xf0 和 0x1c（断码），8042 将其转换成 0x9e 后保存，之后发中断，键盘中断处理程序将其读出，一看是键被弹起，忽略。

过程罗哩罗嗦地说完了，那为什么步骤 4 和步骤 5 不被处理呢？原因是一般情况下，用户的想法是通过

“按”键的方式表达的，强调的是“按”，并不是通过按键松开来表达需求。键被弹起时，一般是用户思想表达完毕的标志，所以步骤 4 和步骤 5 就忽略了，不再考虑 ctrl 键和 a 键的状态。

击键产生的扫描码是由键盘中的 8048 传给主板上的 8042 的，8042 将扫描码转码处理后存入自己的输出缓冲区寄存器中。虽然并不是所有的扫描码都是 1 字节，但它们是以字节为单位发送的（也许这样便于区分按下的是否是那些以 0xe0 开头的多字节操作码，以便后续处理），因此 8042 的输出缓冲区寄存器也是 8 位宽度，即每次只能存储一个扫描码，要么是通码，要么是断码。

只要 8042 收到 1 字节的扫描码后，它就会向中断代理发中断信号。因此按键时所发的中断次数，取决于该键扫描码中包含的字节数，通常情况下键的通码和断码各 1 个，因此通常情况下一个字符会发两次中断，但有的按键的扫描码是多个字节，如右 alt 键，每按一次将产生 4 次中断。可想而知，平时我们打字时，那可是发生了无数次中断啊，对于文字编辑工作来说，中断次数更是天文数字（突然很心疼计算机）。

不知道您注意到没有，表 10-1 中并未列出大写字母，不知道您对此是否感到奇怪，如果觉得这很正常那就我就放心了。前面说过啦，键盘只负责输入，具体的表现形式取决于字符处理软件，虽说一般情况下我们先按下<caps lock>键再按其他字母键时会显示大写字母，但这依然是字符处理软件的功能，只是字符处理软件比较“老实”，它按照约定俗成的规则去表达按键“正常”的行为而已，通常键盘中断处理程序就是充当了字符处理软件的角色，当它收到了<caps lock>键的通码和断码时，也就是表示<caps lock>键已经被按下并弹起，此后便将后面获取的字母的扫描码转换成对应的大写字母，这个转换过程是咱们程序员自己控制的，因此，只要咱们愿意，完全可以无视<caps lock>键，将键入的字母一律展现为小写或大写，甚至展示为数字、标点符号等，当然不建议这么任性，只是想告诉大家，计算机中各种文本，都是字符处理软件的杰作，和键盘关系不大。

总结一下。

- 扫描码有 3 套，现在一般键盘中的 8048 芯片支持的是第二套扫描码。因此每当有击键发生时，8048 发给 8042 的都是第二套键盘扫描码。
- 8042 为了兼容性，将接收到的第二套键盘扫描码转换成第一套扫描码。8042 是按字节来处理的，每处理一个字节的扫描码后，将其存储到自己的输出缓冲区寄存器。
- 然后向中断代理 8059A 发中断信号，这样我们的键盘中断处理程序通过读取 8042 的输出缓冲区寄存器，会获得第一套键盘扫描码。

因此我们可以“假设”现在键盘用的就是第一套扫描键，这样想像比较直接。为了方便对比和“假设”，现将两套扫描码印在同一个键盘上给大伙过目，如图 10-15 所示。

键盘扫描码

▲图 10–15　键盘扫描码示意

好，有关扫描码的部分到这就结束啦，大家下节见。

10.3.3 8042 简介

本节是对 8042 名副其实的简介，因为咱们的键盘操作非常简单，不涉及对其编程，只用了它一个端口接收扫描码而已，所以讲得太细致的话实在是有些矛盾。

说良心话，本节的内容对于咱们的应用来说还是冗余了，原因是我经常有这样的体会：虽然只用到某方面一点点知识，但不把周边内容也介绍的话，这会让人感到迷茫，甚至不知道自己在做什么……好啦，开始开始。

上节中咱们提到了 8042 的输出缓冲区寄存器，这涉及到 8042 的编程，因此咱们先把相关内容介绍下。

考虑到本书中对它的应用实在是有限，甚至是过于有限，介绍多了还“扰民”，因此咱们就浅尝辄止吧。有关 8042 的资料网上就很多呢，大伙有兴趣自己看看。

话得从头说起，其实在计算机中不只一个处理器，我们常说的处理器是指传统意义上的 Intel 或 AMD 的处理器。计算机内部是分层的，各层负责一定的功能，将所有功能串在一起就是一个完整的计算机系统。因此很多外部设备中都有自己的处理器，它们可以响应来自外部的信号和设置硬件本身的功能，最主要的就是它们分担了传统处理器的计算任务，比如显卡的 CPU 称为 GPU，它承担了图像渲染的工作，这样传统处理器就不用做自己不擅长的事。

和键盘相关的芯片只有 8042 和 8048，它们都是独立的处理器，都有自己的寄存器和内存。

Intel 8048 芯片或兼容芯片位于键盘中，它是键盘编码器，相当于键盘的“代言”人，是键盘对外表现击键信息、帮助键盘“说话”的部件。它除了负责监控按键扫描码外，还用来对键盘设置，比如设置键盘上的各种 LED 显示灯的开启和关闭，默认情况下 NumLock 的 LED 灯是亮的，这就是 8048 的功劳。

Intel 8042 芯片或兼容芯片被集成在主板上的南桥芯片中，它是键盘控制器，也就是键盘的 IO 接口，因此它是 8048 的代理，也是前面所得到的处理器和键盘的“中间层”。8048 通过 PS/2、USB 等接口与 8042 通信，处理器通过端口与 8042 通信（IO 接口就是外部硬件的代理，它和处理器都位于主机内部，因此处理器与 IO 接口可以通过端口直接通信）。

既然 8042 是 8048 的 IO 接口，对 8048 的编程也是通过 8042 完成的，所以只要学习 8042 足矣，8048 不再介绍。

8042 有 4 个 8 位的寄存器，如表 10-2 所示。

表 10-2 8042 寄存器

寄 存 器	端 口	读/写
Output Buffer（输出缓冲区）	0x60	读
Input Buffer（输入缓冲区）	0x60	写
Status Register（状态寄存器）	0x64	读
Control Register（控制寄存器）	0x64	写

您看，四个寄存器共用两个端口，这说明在不同场合下同一端口有不同的用途。

8042 是连接 8048 和处理器的桥梁，8042 存在的目的是：为了处理器可以通过它控制 8048 的工作方式，然后让 8048 的工作成果通过 8042 回传给处理器。此时 8042 就相当于数据的缓冲区、中转站，根据数据被发送的方向，8042 的作用分别是输入和输出。

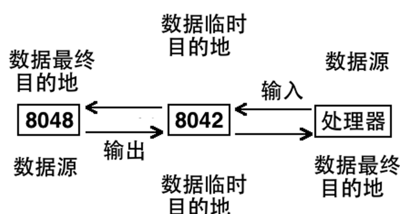
- 处理器把对 8048 的控制命令临时放在 8042 的寄存器中，让 8042 把控制命令发送给 8048，此时 8042 充当了 8048 的参数输入缓冲区。
- 8048 把工作成果临时提交到 8042 的寄存器中，好让处理器能从 8042 的寄存器中获取它（8048）的工作成果，此时 8042 充当了 8048 的结果输出缓冲区。

8042 作为输入、输出缓冲区的区别，如图 10-16 所示。

图 10-16 中，上半部分的数据从左到右传送，表示 8042 作为输入缓冲区，下半部分数据从右到左传送，表示 8042 作为输出缓冲区。

结论：

- 当需要把数据从处理器发到 8042 时（数据传送尚未发生时），0x60 端口的作用是输入缓冲区，此时应该用 out 指令写入 0x60 端口。
- 当数据已从 8048 发到 8042 时，0x60 端口的作用是输出缓冲区，此时应该用 in 指令从 8042 的 0x60 端口（输出缓冲区寄存器）读取 8048 的输出结果。



▲图 10-16　8042 同时作为输入输出缓冲区

下面介绍下各寄存器的作用。

- 输出缓冲区寄存器

8 位宽度的寄存器，只读，键盘驱动程序从此寄存器中通过 in 指令读取来自 8048 的扫描码、来自 8048 的命令应答以及对 8042 本身设置时，8042 自身的应答也从该寄存器中获取。

注意，输出缓冲区寄存器中的扫描码是给处理器准备的，在处理器未读取之前，8042 不会再往此寄存器中存入新的扫描码。

也许您要问了，8042 是怎样知道输出缓冲区寄存器中的值是否被读取了呢？这个简单，8042 也有个智能芯片，它为处理器提供服务，当处理器通过端口跟它要数据的时候它当然知道了，因此，每当有 in 指令来读取此寄存器时，8042 就将状态寄存器中的第 0 位置成 0，这就表示寄存器中的扫描码数据已经被取走，可以继续处理下一个扫描码了。当再次往输出缓冲寄存器存入新的扫描码时，8042 就将状态寄存器中的第 0 位置为 1，这表示输出缓冲寄存器已满，可以读取了。

总之一句话，键盘中断处理程序中必须要用 in 指令读取“输出缓冲寄存器”，否则 8042 无法继续响应键盘操作。

- 输入缓冲区寄存器

8 位宽度的寄存器，只写，键盘驱动程序通过 out 指令向此寄存器写入对 8048 的控制命令、参数等，对于 8042 本身的控制命令也是写入此寄存器。

- 状态寄存器

8 位宽度的寄存器，只读，反映 8048 和 8042 的内部工作状态。各位意义如下。

（1）位 0：置 1 时表示输出缓冲区寄存器已满，处理器通过 in 指令读取后该位自动置 0。

（2）位 1：置 1 时表示输入缓冲区寄存器已满，8042 将值读取后该位自动置 0。

（3）位 2：系统标志位，最初加电时为 0，自检通过后置为 1。

（4）位 3：置 1 时，表示输入缓冲区中的内容是命令，置 0 时，输入缓冲区中的内容是普通数据。

（5）位 4：置 1 时表示键盘启用，置 0 时表示键盘禁用。

（6）位 5：置 1 时表示发送超时。

（7）位 6：置 1 时表示接收超时。

（8）位 7：来自 8048 的数据在奇偶校验时出错。

- 控制寄存器

8 位宽度的寄存器，只写，用于写入命令控制字。每个位都可以设置一种工作方式，意义如下。

（1）位 0：置 1 时启用键盘中断。

（2）位 1：置 1 时启用鼠标中断。

（3）位 2：设置状态寄存器的位 2。

（4）位 3：置 1 时，状态寄存器的位 4 无效。

（5）位 4：置 1 时禁止键盘。

（6）位 5：置 1 时禁止鼠标。

（7）位 6：将第二套键盘扫描码转换为第一套键盘扫描码。

（8）位 7：保留位，默认为 0。

其实还有一些控制命令没说，但咱们真心用不上，甚至连上面介绍的四个寄存器咱们也只用上了一个而已，咱们对 8042 的介绍到此为止。

下一节，我们要实战。

10.3.4 测试键盘中断处理程序

本节通过写一个极其简单的键盘中断处理程序来演示键盘的中断处理过程。

本节中的键盘中断处理程序本身很简单，一会儿您就知道了，其实就两句话的事。写好了的中断处理程序咱得注册到 idt_table 中才行，但现在，我指的是目前，并不是仅仅调用 register_handler 去注册就行了，咱们还有一些前期的准备工作要做。

想想看，咱们的中断机制是这样的，一个中断向量号要有一个中断入口，为了省事，咱们是用宏 VECTOR 来实现的中断入口，因此所有的中断入口程序几乎都一样，只是中断向量号不同。宏展开后，中断入口程序名为 intr%1entry。其中%1 是中断向量号，在入口程序中用这个中断向量号作为 idt_table 中的索引，调用最终 C 语言版本的中断处理程序。

到目前为止，我们只为时钟添加了中断处理程序，它的中断向量号是 0x20。因此在 kernel.S 中，“VECTOR 0x20,ZERO”是最后一个中断入口。

回顾一下 8259A 的 IR 引脚，键盘的中断信号接在主片的 IR1 引脚上，也就是它对应的中断向量为 0x21，因此我们要修改一下 kernel.S，至少添加一句“VECTOR 0x21,ZERO”。

为了一步到位，咱们把 8259A 中的全部中断一次注册好吧，一共 16 个 IR 引脚，咱们在 kernel.S 中再增加 16 个中断入口。请见代码 10-8。

代码 10-8 （project/c10/c/kernel/kernel.S）

```
…略
81 VECTOR 0x20,ZERO        ;时钟中断对应的入口
82 VECTOR 0x21,ZERO        ;键盘中断对应的入口
83 VECTOR 0x22,ZERO        ;级联用的
84 VECTOR 0x23,ZERO        ;串口 2 对应的入口
85 VECTOR 0x24,ZERO        ;串口 1 对应的入口
86 VECTOR 0x25,ZERO        ;并口 2 对应的入口
87 VECTOR 0x26,ZERO        ;软盘对应的入口
88 VECTOR 0x27,ZERO        ;并口 1 对应的入口
89 VECTOR 0x28,ZERO        ;实时时钟对应的入口
90 VECTOR 0x29,ZERO        ;重定向
91 VECTOR 0x2a,ZERO        ;保留
92 VECTOR 0x2b,ZERO        ;保留
93 VECTOR 0x2c,ZERO        ;ps/2 鼠标
94 VECTOR 0x2d,ZERO        ;fpu 浮点单元异常
95 VECTOR 0x2e,ZERO        ;硬盘
96 VECTOR 0x2f,ZERO        ;保留
```

这下主从 8259A 上的 IR 引脚都有相应的中断入口了，在实现系统调用之前够用了。

不过，这才完成了部分工作，咱们还有两个地方要改，它们都是在 interrupt.c 中。

中断入口程序必须在中断描述符表 idt 中注册才行，在中断描述符表中注册中断描述符是在文件 interrupt.c 中用函数 idt_desc_init 实现的，它所注册的中断描述符的数量依赖于 IDT_DESC_CNT，为此我们要把 IDT_DESC_CNT 改为合适的数。

目前咱们已经打开了键盘中断，为了使键盘测试变得简单，咱们暂时先把时钟中断关闭，只打开键盘的中断，您肯定想到了，完成这个目的，就是写 8259A 的中断屏蔽寄存器。

以上两处的修改请见代码 10-9。

代码 10-9 （project/c10/c/kernel/interrup.c）

```
…略
12 #define IDT_DESC_CNT 0x30      // 目前总共支持的中断数
…略
28 static struct gate_desc idt[IDT_DESC_CNT];
// idt 是中断描述符表，本质上就是个中断门描述符数组
29
30 char* intr_name[IDT_DESC_CNT];    // 用于保存异常的名字
31
```

```
32 /********    定义中断处理程序数组    ********
33  * 在 kernel.S 中定义的 intrXXentry 只是中断处理程序的入口，
34  * 最终调用的是 ide_table 中的处理程序*/
35 intr_handler idt_table[IDT_DESC_CNT];
36 /********************************************/
37 extern intr_handler intr_entry_table[IDT_DESC_CNT];
// 声明引用定义在 kernel.S 中的中断处理函数入口数组
38
39 /* 初始化可编程中断控制器 8259A */
40 static void pic_init(void) {
41
42     /* 初始化主片 */
…略
48     /* 初始化从片 */
…略
54   /*测试键盘，只打开键盘中断，其他全部关闭*/
55     outb (PIC_M_DATA, 0xfd);
56     outb (PIC_S_DATA, 0xff);
57
58     put_str("   pic_init done\n");
59 }
…略
```

您看，很多中断相关的结构都与 IDT_DESC_CNT 有关，中断描述符 idt、数组 intr_name 都要用 IDT_DESC_CNT 作为数组长度，甚至所声明的外部数组 intr_entry_table（位于 kernel.S 中）也用到了它，数组 intr_entry_table 中的元素就是程序入口 intr%1entry。

此版本 interrup.c 中的第 12 行将 IDT_DESC_CNT 改为了 0x30，这满足了目前 8259A 所支持的全部外部中断。

第 55 行操作主片上的中断屏蔽寄存器，只打开了键盘中断，即位 1 为 0，其他位都为 1，因此写入的值为 0xfd。

第 56 行操作从片上的中断屏蔽寄存器，屏蔽了从片上的所有中断，所以值为 0xff。

好啦，现在把之前创建的线程也注释掉，现在的 main.c 长这样，见代码 10-10。

代码 10-10　（project/c10/c/kernel/main.c）

```
10 int main(void) {
11    put_str("I am kernel\n");
12    init_all();
13
14 //   thread_start("k_thread_a", 31, k_thread_a, "argA ");
15 //   thread_start("k_thread_b", 8, k_thread_b, "argB ");
16
17    intr_enable();
18    while(1); //{
19       //console_put_str("Main ");
20   // };
21    return 0;
22 }
```

做了这么多，我们只是在做准备工作而已，咱们的键盘中断处理程序还没写呢。不要急，很简单的。

我们的设备驱动文件都放在 device 目录中，为此，我们在该目录下创建文件 keyboard.c，在其中定义键盘中断处理程序。给各位客官上菜喽，见代码 10-11。

代码 10-11　（project/c10/c/device/keyboard.c.10-10）

```
 1 #include "keyboard.h"
 2 #include "print.h"
 3 #include "interrupt.h"
 4 #include "io.h"
 5 #include "global.h"
 6
 7 #define KBD_BUF_PORT 0x60   // 键盘 buffer 寄存器端口号为 0x60
 8
 9 /* 键盘中断处理程序 */
10 static void intr_keyboard_handler(void) {
11    put_char('k');
```

```
12 /* 必须要读取输出缓冲区寄存器，否则 8042 不再继续响应键盘中断 */
13     inb(KBD_BUF_PORT);
14     return;
15 }
16
17 /* 键盘初始化 */
18 void keyboard_init() {
19     put_str("keyboard init start\n");
20     register_handler(0x21, intr_keyboard_handler);
21     put_str("keyboard init done\n");
22 }
```

代码 10-10 是不是太短小精悍了？至少它能很好地演示键盘中断机制。其中函数 intr_keyboard_handler 就是键盘中断处理程序，它的实现就两句话，每收到一个中断，就通过 put_char('k')打印字符'k'，然后再调用 inb(KBD_BUF_PORT)读取 8042 的输出缓冲区寄存器。

顺便说一句，函数 inb 是有返回值的，它返回的是从端口读取的数据，虽然此处没有将其赋给任何变量，但不要觉得这会让返回值“没地方放”，然后整个人都觉得不好了，因为根据 ABI 约定，返回值是存放在寄存器 eax 中的，我们这里没有将返回值赋给内存变量，编译器只是没有将返回值从 eax 寄存器中 mov 到某块内存而已。

这个例子有两个目的，一是为了演示击键时产生的中断次数，每按一下键，您看产生多少个字符'k'就行了。二是为了演示不读取输出缓冲区寄存器的话，8042 是不会继续工作的。

编译运行，咱们在 bochs 中测试一下，如图 10-17 所示。

注意，咱们这次要在 bochs 的屏幕窗口中按键，不是在控制台中输入，否则就会被当成调试命令。

```
thread_init start
thread_init done
timer_init start
timer_init done
keyboard init start
keyboard init done
kkkkkk
```

▲图 10-17　键盘中断测试

当我在屏幕窗口中输入 a 键时，由于 a 的通码和断码各为一字节，屏幕上打印出两个 k，左边有下画线的那一对 k，前一个 k 代表通码的中断，后一个 k 代表的是断码的中断。当我再次按下<R-alt>键时，其通码和断码各为 2 字节，因此产生了 4 次中断，故打印了 4 个字符 k。

```
thread_init start
thread_init done
timer_init start
timer_init done
keyboard init start
keyboard init done
k_
IPS: 30.878M          NUM   CAPS
```

▲图 10-18　未读取输出缓冲区寄存器时

当我注释掉“inb(KBD_BUF_PORT)”后，由于 8042 的输出缓冲区寄存器未被读取，任凭我怎么敲键盘，屏幕窗口都只显示了一个字符 k，8042 不再处理任何按键信息，如图 10-18 所示。

现在您“领教”了不读取输出缓冲区寄存器的厉害了？哈哈，我想您还是对扫描码比较好奇，扫描码必须要读取“输出缓冲区寄存器”才能得到，现在改一下代码，如代码 10-12 所示。

代码 10-12　（project/c10/c/device/keyboard.c.10-11）

```
…略
 9 /* 键盘中断处理程序 */
10 static void intr_keyboard_handler(void) {
11 /* 必须要读取输出缓冲区寄存器，否则 8042 不再继续响应键盘中断 */
12     uint8_t scancode = inb(KBD_BUF_PORT);
13     put_int(scancode);
14     return;
15 }
…略
```

编译运行后，在屏幕上继续按 a 键和<R-alt>，结果如图 10-19 所示。

咱们还是看有下画线的两组数据。左边第一组中，0x1E 为 a 键的通码，0x9E 为 a 键的断码。第二组中，0xE0 和 0x38 是<R-alt>的通码，它的断码为 0xE0 和 0xB8。

简单演示就到这了，下一节中咱们让键盘操作“正常”起来，按什么就显示什么，像通常的键盘输入那样直观。

```
mem_init done
thread_init start
thread_init done
timer_init start
timer_init done
keyboard init start
keyboard init done
1E9EE038E0B8
```

▲图 10-19　打印扫描码

10.4 编写键盘驱动

驱动程序是什么？

还是那句话，计算机按功能分层，各层模块各司其职，下层模块为上层提供服务，这里的模块我指的是硬件。操作系统看似功能强大，但它的能力取决于硬件能做什么。

在计算机中，硬件是用软件来交互的，想让硬件做什么，必须通过软件的方式告诉它。硬件为方便软件对它的“调遣”，它为软件提供了接口，这通常是通过 IO 指令进行一堆复杂的寄存器设置，然后通过读取寄存器检测相应的状态，然后再进行数据交换。虽然这已大大方便了我们对硬件的控制，但我们依然是“懒惰的”，不希望每次找硬件帮忙时都做这种重复性的体力劳动，这种很“直白地”寄存器控制指令显然还方便得不够。我们不想要“过程”，只想要个“结果”。

为了方便获取“结果”，我们将这些复杂的硬件控制指令封装成一个过程，每次只把对硬件的操作需求提交给此过程，由此过程实施底层的控制细节，然后返回给调用者一个结果，这个直接同底层硬件打交道的过程便是驱动程序。

今天我们要编写键盘中断处理程序，这是我们的第一个硬件驱动程序。

10.4.1 转义字符介绍

在开始动手写代码之前，先介绍点转义字符的知识，也许您用得着。

字符集中的字符分为两大类，一类是可见字符，如字符'w'，这是看得见的字符。另一类是不可见的控制字符，比如回车符、制表符等。字符通常情况下是通过键盘输入的（手写板、鼠标点击都不算），因此，键盘中的各种键也分为两大类，一类按键负责输入可见字符，另一类按键负责输入控制字符。

前面介绍过，按键的行为是由字符处理软件来解释的，字符处理软件通常是上层模块，它所处理的对象并不是扫描码，而是字符集中的字符编码，如 ASCII 码。因此，我们的键盘驱动有责任将扫描码转换成对应的 ASCII 码。

转换工作就是建立源到目标映射关系，因此几乎都是硬编码，也就是这种映射关系是在程序中固定写死的，比如当扫描码为 0x2 时（通码），在未按住 shift 键的情况下，我们将其转换为字符'1'。

于是问题来了，键盘上的控制键，它们对应的字符是不可见的，对于这些不可见的字符，我们对其转换的时候，如何在程序中写入对应的字符呢？聪明的你肯定想到了转义字符。

大伙儿一般用到的转义字符都是“反斜杠字符'\'开头+单个字母”的形式，比如水平制表符用 Tab 键用'\t'表示，退格键 backspace 用'\b'表示，但有些字符并没有字符表示的形式，比如 Esc 键。

不知道大伙是否都了解，在 C 语言中有三种转义字符。

（1）一般转义字符，'\+单个字母'的形式。

（2）八进制转义字符，'\0+三位八进制数字表示的 ASCII 码'的形式。

（3）十六进制转义字符，'\x+两位十六进制数字表示的 ASCII 码'的形式。

上面的'\t'就属于第 1 种一般转义字符。因此若某些控制键没有一般转义字符，咱们可以用八进制或十六进制转义字符去表示它，总之后两种是表示任意字符的万能方法，您甚至可以在字符串中用夹杂这种转义形式表示字符，屡试不爽。拿字符'&'的 ASCII 码举例，十进制为 38，八进制为 46，十六进制为 0x26，下面用 echo 命令做了演示，如图 10-20 所示。

```
[work@localhost tmp]$ echo -en 'u\046me\n'
u&me
[work@localhost tmp]$ echo -en 'u\x26me\n'
u&me
[work@localhost tmp]$
```

▲图 10-20　八进制和十六进制转义字符

您看，对于字符'&'的 ASCII 码，第一条 echo 命令是用八进制转义的，第二条 echo 命令是用十六进制转义的，效果相同，都打印出亲情般温馨的 u&me。

10.4.2 处理扫描码

想必通过之前的例子，大伙儿已经了解键盘输入的本质了。键盘中断处理程序负责接收按键信息，也就是按键的扫描码，然后就是对各种扫描码的处理，把不同扫描码解释为不同的表现行为。简单来说这也是驱动程序做的事，但只是一部分，本节要完成这部分工作。

分析一下人家都是怎么处理的，通常情况下：

当按下的键是可见字符时，屏幕上都会将其显示出来，比如按下了 a 键，屏幕上应该输出字符'a'，给用户一个反馈，这样用户才觉得自己没按错，这是为打造好的用户体验最起码的素质。

当按下的键是控制字符时，我们应该做出相应的控制行为，并在屏幕上展现出这种行为，比如按下了 backspace 键，咱们也应该在屏幕上让用户觉得光标所在处前面的字符被删掉了。

咱们也按照这种传统的作法实现，这分两阶段来完成。

（1）如果是一些用于操作方面的控制键，简称操作控制键，如<shift>、<ctrl>、<caps lock>，它通常是组合键，需要与其他键一起考虑，然后做出具体的行为展现，在键盘驱动中完成处理。

（2）如果是一些用于字符方面的键，无论是可见字符，或是字符方面的控制键（简称字符控制键），如<backspace>，统统交给字符处理程序完成，比如咱们的 put_char。还记得吗？put_char 能够处理<backspace>，也就是'\b'。

对于第一阶段，它与字符无直接的关系，因此咱们就在键盘驱动中处理。

对于第二阶段，咱们得知道用户按下的是什么字符，不能把操作控制键当成字符传给字符处理程序，比如把 shift 键的扫描码传给 put_char，这不就乱了吗？因此，咱们得把按键的扫描码转换成对应的字符，也就是将通码转换为字符的 ASCII 码，这就是前面所说的源到目标的映射关系。

大伙儿看下表 10-1，虽然表中的键看似无规律，但仔细观察一下，它们的通码几乎是连续的，范围是 0x1～0x58，其中 0x54～0x56 不存在，其他键的通码都是连续分布的。因此，咱们可以创建二维数组来记录这映射关系，用通码作为数组的索引。

目前咱们用不到所有键的功能，为处理简单，这里并不打算支持所有的按键，暂时只支持主键盘区，因此咱们的数组范围暂时为 0x1～0x3A，以后用到的时候再加吧。

好啦，可以介绍键盘中断处理程序了，请见代码 10-13-1。

代码 10-13-1 （project/c10/d/device/keyboard.c）

```
 1 #include "keyboard.h"
 2 #include "print.h"
 3 #include "interrupt.h"
 4 #include "io.h"
 5 #include "global.h"
 6
 7 #define KBD_BUF_PORT 0x60    // 键盘 buffer 寄存器端口号为 0x60
 8
 9 /* 用转义字符定义部分控制字符 */
10 #define esc        '\033'    // 八进制表示字符，也可以用十六进制'\x1b'
11 #define backspace   '\b'
12 #define tab        '\t'
13 #define enter      '\r'
14 #define delete     '\177'   // 八进制表示字符，十六进制为'\x7f'
15
16 /* 以上不可见字符一律定义为 0 */
17 #define char_invisible  0
18 #define ctrl_l_char     char_invisible
19 #define ctrl_r_char     char_invisible
20 #define shift_l_char    char_invisible
21 #define shift_r_char    char_invisible
22 #define alt_l_char      char_invisible
23 #define alt_r_char      char_invisible
24 #define caps_lock_char  char_invisible
25
26 /* 定义控制字符的通码和断码 */
27 #define shift_l_make   0x2a
```

```
28 #define shift_r_make    0x36
29 #define alt_l_make     0x38
30 #define alt_r_make     0xe038
31 #define alt_r_break    0xe0b8
32 #define ctrl_l_make    0x1d
33 #define ctrl_r_make    0xe01d
34 #define ctrl_r_break   0xe09d
35 #define caps_lock_make  0x3a
36
37 /* 定义以下变量记录相应键是否按下的状态，
38  * ext_scancode 用于记录 makecode 是否以 0xe0 开头 */
39 static bool ctrl_status, shift_status, alt_status, caps_lock_status, ext_scancode;
40
```

代码 10-12-1 只是 keyboard.c 的一部分，程序开头定义了 KBD_BUF_PORT 为 0x60，这是 8042 输入和输出缓冲区寄存器的端口。下面是预先定义的按键扫描码，都是用宏定义的控制键的通码（make）及断码（break），它们将在扫描码数组中用到。

第 10～14 行定义了控制键的 ASCII 码，由于它们都是不可见字符，它们都是用转义字符的形式定义的。其中 esc 和 delete 没有一般转义字符的形式，因此只能考虑八进制或十六进制形式。您看，咱们这里用的是八进制形式，这么做主要是为了兼容，并不是所有的 C 标准都支持这两种数字形式的转义字符，八进制出现的较早，很古老的编译器也支持，但十六进制形式的转义字符是在 c89 后才实现的，尽管目前 gcc 都支持 c89，甚至有的编译器都支持 c99 了，但保持兼容也没什么不好。

第 17～24 行定义的是操作控制键的“ASCII 码”，加了引号的原因您懂的，这类键是没有 ASCII 的，在控制键中只有字符控制键才有 ASCII 码。后面定义的数组中要用到 ASCII 码，为了与数组保持格式上的一致，所以将其统一定义为 0，相当于占位用的，一会儿介绍数组时您就清楚了。

第 27～35 行定义的是操作控制键的扫描码，它们是前面提到过的“两阶段”中的第一阶段，用于组合键中的判断。比如按下了 shift 时再按字母键，就表示输入的是大写字符。

操作控制键与其他键配合时是先被按下的，因此，每次在接收一个按键时，需要查看上一次是否有按下相关的操作控制键，所以咱们得记录操作控制键在之前是否被按下了，也就是将操作控制键的当前状态记录在某个全局变量中，我们在第 39 行声明的全局变量就是用于这个目的的。

下面咱们继续看代码 10-13-2，还是 keyboard.c。

代码 10-13-2　（project/c10/d/device/keyboard.c）

```
…略
 41 /* 以通码 make_code 为索引的二维数组 */
 42 static char keymap[][2] = {
 43 /* 扫描码未与 shift 组合*/
 44 /* ---------------------------------- */
 45 /* 0x00 */     {0,    0},
 46 /* 0x01 */     {esc,   esc},
 47 /* 0x02 */     {'1',   '!'},
 48 /* 0x03 */     {'2',   '@'},
 49 /* 0x04 */     {'3',   '#'},
 50 /* 0x05 */     {'4',   '$'},
 51 /* 0x06 */     {'5',   '%'},
 52 /* 0x07 */     {'6',   '^'},
 53 /* 0x08 */     {'7',   '&'},
 54 /* 0x09 */     {'8',   '*'},
 55 /* 0x0A */     {'9',   '('),
 56 /* 0x0B */     {'0',   '}'},
 57 /* 0x0C */     {'-',   '_'},
 58 /* 0x0D */     {'=',   '+'},
 59 /* 0x0E */     {backspace, backspace},
 60 /* 0x0F */     {tab,   tab},
 61 /* 0x10 */     {'q',   'Q'},
 62 /* 0x11 */     {'w',  'W'},
 63 /* 0x12 */     {'e',   'E'},
 64 /* 0x13 */     {'r',   'R'},
 65 /* 0x14 */     {'t',   'T'},
 66 /* 0x15 */     {'y',   'Y'},
 67 /* 0x16 */     {'u',   'U'},
```

```
 68 /* 0x17 */     {'i',   'I'},
 69 /* 0x18 */     {'o',   'O'},
 70 /* 0x19 */     {'p',   'P'},
 71 /* 0x1A */     {'[',   '{'},
 72 /* 0x1B */     {'}',   '}'},
 73 /* 0x1C */     {enter,  enter},
 74 /* 0x1D */     {ctrl_l_char, ctrl_l_char},
 75 /* 0x1E */     {'a',   'A'},
 76 /* 0x1F */     {'s',   'S'},
 77 /* 0x20 */     {'d',   'D'},
 78 /* 0x21 */     {'f',   'F'},
 79 /* 0x22 */     {'g',   'G'},
 80 /* 0x23 */     {'h',   'H'},
 81 /* 0x24 */     {'j',   'J'},
 82 /* 0x25 */     {'k',   'K'},
 83 /* 0x26 */     {'l',   'L'},
 84 /* 0x27 */     {';',   ':'},
 85 /* 0x28 */     {'\'', '"'},
 86 /* 0x29 */     {'`',   '~'},
 87 /* 0x2A */     {shift_l_char, shift_l_char},
 88 /* 0x2B */     {'\\',  '|'},
 89 /* 0x2C */     {'z',   'Z'},
 90 /* 0x2D */     {'x',   'X'},
 91 /* 0x2E */     {'c',   'C'},
 92 /* 0x2F */     {'v',   'V'},
 93 /* 0x30 */     {'b',   'B'},
 94 /* 0x31 */     {'n',   'N'},
 95 /* 0x32 */     {'m',   'M'},
 96 /* 0x33 */     {',',   '<'},
 97 /* 0x34 */     {'.',   '>'},
 98 /* 0x35 */     {'/',   '?'},
 99 /* 0x36 */     {shift_r_char, shift_r_char},
100 /* 0x37 */     {'*',   '*'},
101 /* 0x38 */     {alt_l_char, alt_l_char},
102 /* 0x39 */     {' ',   ' '},
103 /* 0x3A */     {caps_lock_char, caps_lock_char}
104 /*其他按键暂不处理*/
105 };
106
```

这里定义的二维数组 keymap 便是本节键盘驱动的核心，此数组主要是定义了与 shift 组合时的字符效果。数组范围是 0～0x3A，这是咱们说好的目前所支持的主键盘区的按键范围，您可以对照图 10-15 中的第一套按键扫描码看看。

主键盘区主要是与上档键 shift 配合使用，您看，如果之前已经按下了 shift 键并且按住不松手：

- 当在主键盘区中按下数字键时，这表示按键为数字上面的符号，如'3'变成了'#'。
- 当在主键盘区中按下字母键时，这表示按键为大写字母，如'a'变成了'A'。

有无 shift 键参与时按键效果是不同的，所以我们主要以这种和 shift 键配合的情况来建立通码到对应字符的映射，这里的字符就是指字符的 ASCII 码。

keymap 中每个数组元素都是一维数组，代表某个按键在有无 shift 键配合情况下的表现。

一维数组中的第 0 个元素是某个按键未与 shift 键组合时对应的字符 ASCII 码值，第 1 个元素是某个按键与 shift 键组合时对应的字符 ASCII 码值。举个例子，a 的通码为 0x1e，比如按下 a 键，若之前未按下 shift 键，咱们应该处理为小写字符'a'，所以 keymap[0x1e][0]等于'a'。否则若已经按下了 shift 键，咱们应该处理为大写字符'A'，所以 keymap[0x1e][1]等于'A'。

没有通码为 0 的键，因此数组第 0 个一维数组 keymap[0][]，咱们为其定义两个 0 值。

keymap 数组就介绍到这，下面咱们看代码 10-13-3。

代码 10-13-3　（project/c10/d/device/keyboard.c）

```
…略
107 /* 键盘中断处理程序 */
108 static void intr_keyboard_handler(void) {
109
110 /* 这次中断发生前的上一次中断，以下任意三个键是否有按下 */
```

```
111     bool ctrl_down_last = ctrl_status;
112     bool shift_down_last = shift_status;
113     bool caps_lock_last = caps_lock_status;
114
115     bool break_code;
116     uint16_t scancode = inb(KBD_BUF_PORT);
117
118 /* 若扫描码 scancode 是 e0 开头的，表示此键的按下将产生多个扫描码，
119   * 所以马上结束此次中断处理函数，等待下一个扫描码进来*/
120     if (scancode == 0xe0) {
121         ext_scancode = true;    // 打开 e0 标记
122         return;
123     }
124
125 /* 如果上次是以 0xe0 开头的，将扫描码合并 */
126     if (ext_scancode) {
127         scancode = ((0xe000) | scancode);
128         ext_scancode = false;   // 关闭 e0 标记
129     }
130
131     break_code = ((scancode & 0x0080) != 0);   // 获取 break_code
132
133     if (break_code) { // 若是断码 break_code（按键弹起时产生的扫描码）
134
135     /* 由于 ctrl_r 和 alt_r 的 make_code 和 break_code 都是两字节，
136     所以可用下面的方法取 make_code，多字节的扫描码暂不处理 */
137        uint16_t make_code = (scancode &= 0xff7f);
           // 得到其 make_code（按键按下时产生的扫描码）
138
139     /* 若是任意以下三个键弹起了，将状态置为 false */
140        if (make_code == ctrl_l_make || make_code == ctrl_r_make) {
141            ctrl_status = false;
142        } else if (make_code == shift_l_make || make_code == shift_r_make) {
143            shift_status = false;
144        } else if (make_code == alt_l_make || make_code == alt_r_make) {
145            alt_status = false;
146        } /* 由于 caps_lock 不是弹起后关闭，所以需要单独处理 */
147
148        return;   // 直接返回结束此次中断处理程序
149
150     }
151 /* 若为通码，只处理数组中定义的键以及 alt_right 和 ctrl 键，全是 make_code */
152     else if ((scancode > 0x00 && scancode < 0x3b) || \
153                   (scancode == alt_r_make) || \
154                   (scancode == ctrl_r_make)) {
155        bool shift = false;
                  // 判断是否与 shift 组合，用来在一维数组中索引对应的字符

156        if ((scancode < 0x0e) || (scancode == 0x29) || \
157            (scancode == 0x1a) || (scancode == 0x1b) || \
158            (scancode == 0x2b) || (scancode == 0x27) || \
159            (scancode == 0x28) || (scancode == 0x33) || \
160            (scancode == 0x34) || (scancode == 0x35)) {
161                /****** 代表两个字母的键 ********
162                          0x0e 数字'0'～'9',字符'-',字符'='
163                          0x29 字符'`'
164                          0x1a 字符'['
165                          0x1b 字符']'
166                          0x2b 字符'\\'
167                          0x27 字符';'
168                          0x28 字符'\''
169                          0x33 字符','
170                          0x34 字符'.'
171                          0x35 字符'/'
172                *******************************/
173            if (shift_down_last) {  // 如果同时按下了 shift 键
174                shift = true;
175            }
176        } else {    // 默认为字母键
177            if (shift_down_last && caps_lock_last) {
                                       // 如果 shift 和 capslock 同时按下
178                shift = false;
```

```
179             } else if (shift_down_last || caps_lock_last) {
                                         // 如果 shift 和 capslock 任意被按下
180                 shift = true;
181             } else {
182                 shift = false;
183             }
184         }
185
186         uint8_t index = (scancode &= 0x00ff);
            // 将扫描码的高字节置 0，主要针对高字节是 e0 的扫描码

187         char cur_char = keymap[index][shift];  // 在数组中找到对应的字符
188
189         /* 只处理 ASCII 码不为 0 的键 */
190         if (cur_char) {
191             put_char(cur_char);
192             return;
193         }
194
195  /* 记录本次是否按下了下面几类控制键之一，供下次键入时判断组合键 */
196         if (scancode == ctrl_l_make || scancode == ctrl_r_make) {
197             ctrl_status = true;
198         } else if (scancode == shift_l_make || scancode == shift_r_make) {
199             shift_status = true;
200         } else if (scancode == alt_l_make || scancode == alt_r_make) {
201             alt_status = true;
202         } else if (scancode == caps_lock_make) {
203         /* 不管之前是否有按下 caps_lock 键，当再次按下时则状态取反，
204          * 即已经开启时，再按下同样的键是关闭。关闭时按下表示开启*/
205           caps_lock_status = !caps_lock_status;
206         }
207      } else {
208         put_str("unknown key\n");
209      }
210 }
211
…略
```

代码 10-13-3 确实有点长，键盘处理程序相对来说是个“大活儿”，拆开了不好说，因此一股脑就全塞过来了，咱们边看边介绍。

键盘中断处理程序始终是每次处理一个字节，所以当扫描码中是多字节时，或者有组合键时，咱们要定义额外的全局变量来记录它们曾经被按下过。

ctrl_status、shift_status 和 caps_lock_status 是定义在代码 10-12-1 结尾处的三个全局变量，它们分别记录<ctrl>、<shift>和<caps lock>三个键的状态，值为 true 表示按下，值为 false 表示弹起。每次这三个键被按下或被弹起时，都将记录在这些变量中，对这三个键的处理是在 intr_keyboard_handler 中对通码和断码各自的处理代码块的结尾处，一会儿咱们再说。

您看，我们只定义了 ctrl_status、shift_status、alt_status 和 caps_lock_status 这 4 个控制键，其中用于组合键的只有前三个，因此，咱们最多支持<ctrl>+<shift>+<alt>三个控制键形式的组合键。

函数 intr_keyboard_handler 不再像上一版本那么简单，这次显得像样一些了。在程序开头定义了三个布尔变量 ctrl_down_last、shift_down_last 和 caps_lock_last，分别从那三个全局变量中获取这三个键曾经是否被按下并且尚未松开。

在第 116 行从端口 KBD_BUF_PORT 获取扫描码，开始处理。

咱们目前只支持主键盘区上的键，因此只存在一个 0xe0 作为扫描码前缀的情况。我们这里用变量 ext_scancode 作为 0xe0 扩展扫描码的标记。

在第 120 行只要发现扫描码为 0xe0，就表示此键的扫描码多于一个字节，后面还有扫描码，因此将 ext_scancode 标记置为 true 后执行 return 返回。

在第 126 行通过 ext_scancode 标记判断上一次是否收到了 0xe0，如果是，将此次接收到的扫描码与 0xe0 合并为完整的扫描码（此时为通码或断码）用于后续处理，并且将 ext_scancode 置为 false，关闭扩展扫描码标记。

现在不知道此次接收的扫描码是通码，还是断码，也就是不知道现在是按键按下，还是按键弹起，因此在第 131 行通过 break_code = ((scancode & 0x0080) != 0)来判断扫描码是否为断码。断码的第 8 位为 1，所以用扫描码 scancode 和 0x0080 进行位与操作，此时 bread_code 的值为 true 或 false。

第 133 行判断若为断码，就进入断码的处理代码块中。

接下来我们要判断此次按键（扫描码）对应的字符是什么。因为我们的扫描码对应的字符定义在二维数组 keymap 中，通码是此数组的索引，所以此时接收的若为断码，为了检索数组 keymap，我们还是要将其还原为通码。

大伙儿知道，第一套键盘扫描码通码和断码的区别就是扫描码第 8 位的值，断码的第 8 位为 1，通码的第 8 位为 0。因此，在第 137 行，将扫描码 scancode（此时它为断码）与 0xff7f 进行位与运算，抹去第 8 位的 1，这样就获得了其通码，并将其存储到变量 make_code 中。

现在我们已经获得了断码对应的通码，下面我们就用“通码”来处理键盘的“弹起”，加引号的目的是想强调，咱们还是在断码的处理流程中，处理的是按键的弹起，并不是按下，只是用通码更为方便判断是哪个按键。

第 140～146 行判断此通码是否为 ctrl、shift、alt。一般情况下这三个键在键盘上都是左右各一个，所以无论按下哪个都表示按下了同一功能的控制键，因此无论弹起哪个也都表示弹起了同一功能的控制键。

拿第 140 行的 ctrl 键的判断来说：“if (make_code == ctrl_l_make || make_code == ctrl_r_make)”，拿通码 make_code 分别与左 ctrl 键的通码 ctrl_l_make 和右 ctrl 键的通码 ctrl_r_make 比较，若满足其一，便将 ctrl_status 置为 false，表示 ctrl 键此时被松开弹起了。注意是置为 false，虽然我们是用通码在判断，但现在处于处理断码的代码块中，此代码块就是判断是哪个键被弹起了。

前面咱们分析过组合键的弹起顺序了，一般是先松开控制键，再松开字符键，所以这三个键的状态变量 ctrl_status、shift_status 和 alt_status 并不是本次使用，是供下次判断组合键用的，本次只是记录是否松开了它们。下次在进入键盘中断时，在 intr_keyboard_handler 的开头通过 ctrl_down_last = ctrl_status 获取上一次 ctrl 键是否处于弹起的状态（也就是没有按下它）。

对于 alt 和 shift 的处理也是一样，最后结束断码的处理，通过 return 返回。

以上是对断码的处理。

若此次接收的键为通码的话，则进入通码的处理代码块，这里的主要工作就是根据通码和 shift 键是否按下的情况，在数组 keymap 中找到按键对应的字符。

我们最大支持的通码为 0x3a，即只支持到<caps_lock>键，因此咱们要防止越界，所以第 152～154 行通过“scancode > 0x00 && scancode < 0x3b”来限制对数组 keymap 的访问。另外，我们将来也要支持 ctrl 或 alt 相关的快捷键，但<R-ctrl>和<R-alt>的通码是以 0xe0 开头的扩展扫描码，范围不在 0x3b 之内，所以加了“或”判断“(scancode == alt_r_make) || (scancode == ctrl_r_make)”。

接下来我们要将扫描码转换为字符了。

主键盘区主要就是数字键和字母键，因此现在要考虑的是之前是否按下了<shift>键和<capslock>键，大伙儿知道，主键盘区中部分键有两个意义，当与 shift 配合使用时，表示键中上面的字符，为方便讨论，暂称它们为双字符键。<shift>键和<capslock>键对双字符键和字母键的影响是不一样的，下面分别讨论。

- 当键入的是双字符键时

如果同时按下了<shift>键，则应该转换为数字键上面的那个符号，比如当前按下的是数字 2，之前若按下 shift 未松手的话，现在应该将其转换为字符'@'。

<capslock>键是否开启，对双字符键无影响。

- 当键入的是字母键时

如果之前开启了<caps lock>键，则应该转换为大写字母。

如果之前同时按下了<shift>键不松手，但没有按下<capslock>键，则也应该转换为大写字母。

若之前同时按下了<capslock>键和<shift>键，<shift>键将<capslock>键的功能抵消，因此键入的是字母键应该转换为小写字母。

实际上，转换后的具体字符是在二维数组 keymap 中的一维数组中，以上的讨论本质上是决定使用按

键所在一维数组中的第 0 个元素，还是第 1 个元素作为转换后的字符。第 0 个元素是无<shift>键时对应的字符，第 1 个元素是有<shift>键时对应的字符，无论<capslock>键和<shift>键怎样配合，其最终效果等同于<shift>键要么按下，要么松开。因此，我们可以把<capslock>键和<shift>键的各种配合统统归并于<shift>键是否按下，我们用个 bool 变量 shift 来记录<shift>键的状态，依然是值为 1 表示按下，值为 0 表示弹起，因此，咱们目前只要确定 shift 的值便确定了转换结果，总结为：

- 若 shift 为 false，则表示 shift 为 0，这表示一维数组中第 0 个元素，即 keymap[通码][0]。
- 若 shift 为 true，则表示 shift 为 1，这表示一维数组中第 1 个元素，即 keymap[通码][1]。

第 155 行定义了布尔变量 shift，用它来索引按键所在一维数组中的两个元素，默认为 false，即 bool shift = false。

下面分两种情况来处理，先处理双字符键。

双字符键也不少，下面列出它们的通码及字符意义。

- 数字 0～9、字符'-'、字符'='的通码是< 0x0e。
- 字符'`'的通码是 0x29。
- 字符'['的通码是 0x1a。
- 字符']'的通码是 0x1b。
- 字符'\\'的通码是 0x2b。
- 字符';'的通码是 0x27。
- 字符'\''的通码是 0x28。
- 字符','的通码是 0x33。
- 字符'.'的通码是 0x34。
- 字符'/'的通码是 0x35。

第 156～160 行的代码便是对它们的判断。

如果是这些双字符键，并且按下了<shift>，也就是 if (shift_down_last)成立的话，就将 shift 置为 true。

下面处理字母键，主键盘区中的可见字符，除了双字符键就是字母键。

“if (shift_down_last && caps_lock_last)”表示如果同时按下<shift>和<caps lock>，抵销了大写功能，所以 shift 为 false。

代码“else if (shift_down_last || caps_lock_last)”表示如果<shift>和<caps lock>仅按下了一个，大写功能开启，因此 shift 为 true。

在其他情况下，shift 为 false。

接下来我们要用通码作为数组 keymap 的索引了，在第 186 行声明变量 index 来记录数组索引值，此时扫描码为通码，直接拿扫描码和 0x00ff 做位与操作就可以了，即代码 uint8_t index = (scancode &= 0x00ff)。为什么用 0x00ff，而不用 0xff？原因是有可能此扫描码是 2 字节，高字节是 0xe0，我们要抹去它。

Index 作为二级数组 keymap 中的索引，shift 作为二级数组 keymap 中一级数组的索引，需要的数据齐了，所以下一行通过代码“cur_char = keymap[index][shift]”在数组中找到对应的字符。

在数组 keymap 中，部分控制字符（如操作控制键）是不可见的，其值为 0，我们没法显示它们，所以在第 190 行通过“if (cur_char)”来判断 cur_char 是否为 0，若不为 0，则表示当前按键是可显示字符或格式控制字符，然后通过“put_char(cur_char)”将其输出，随后通过 return 返回。

如果 cur_char 为 0，根据目前 keymap 的定义，表示它们是操作控制键<ctrl>、<shift>、<alt>或<capslock>之一，只有这 4 个按键对应的 ASCII 码为 0，注意，现在处于通码的代码块中，因此要判断下是它们中的哪一个按下了。

接下来便是对这 4 个键进行判断，如果哪个键被按下了就将其状态变量置为 true，供下次判断组合键。举个例子，比如本次按下了 ctrl 键（无论是<L-ctrl>，还是<R-ctrl>），这句判断“if (scancode == ctrl_l_make || scancode == ctrl_r_make)”都会成立，因此就会将状态变量 ctrl_status 置为 true，即 ctrl_status = true。之后中断执行完毕，退出。

前面咱们分析过组合键的按下顺序了，一般是先按下控制键，再按下字符键，所以这三个键的状态变

量 ctrl_status、shift_status 和 alt_status 并不是本次使用，是供下次判断组合键用的，本次只是记录是否按下了它们。下次在进入键盘中断时，在 intr_keyboard_handler 的开头通过 ctrl_down_last = ctrl_status 获取上一次 ctrl 键是否按下。

<caps lock>键有些特殊，它不像<ctrl>、<shift>、<alt>那样“按下时”开启，“弹起时”关闭。<caps lock>生效于每一次完整的击键操作，即“按下再弹起”开启，下一次的“按下再弹起”则关闭，相当于状态取反，开启时按下此键是关闭，关闭时按下此键是开启。因此对它的处理也特别一点，对此键的状态取反即可，即代码 caps_lock_status = !caps_lock_status。

咱们支持的键是有限的，对于那些通码在 0x3b 以上的按键，咱们在 else 代码块中执行 put_str("unknown key\n")提示未知键。

键盘驱动就介绍完了，下面编译运行看效果。

此次编译时 gcc 提示警告变量 ctrl_down_last 未用到，这个变量以后会用，暂时忽略警告，如图 10-21 所示。

好久没编译了，这里是通过 make build 1>/dev/null 来编译的，咱们通常只关注报错，因此这里将标准输出重定向到/dev/null，这样就不会显示多余的编译信息扰乱咱们了。最后通过 make hd 写入硬盘，当然也可以直接执行 make all。

在 bochs 中运行效果如图 10-22 所示。

```
[work@localhost d]$ make build 1>/dev/null
device/keyboard.c: 在函数‘intr_keyboard_handler’中：
device/keyboard.c:111: 警告：未使用的变量‘ctrl_down_last’
[work@localhost d]$ make hd
dd if=./build/kernel.bin \
          of=/home/work/my_workspace/bochs/hd60M.img \
          bs=512 count=200 seek=9 conv=notrunc
记录了41+1 的读入
记录了41+1 的写出
21088字节(21 kB)已复制，0.000140782 秒，150 MB/秒
[work@localhost d]$
```

▲图 10-21　编译、警告、写入硬盘

```
keyboard init start
keyboard init done

abcdeFGHIjklm
It works! *^_^*
IPS: 30.705M    NUM
```

▲图 10-22　键盘驱动

在 keyboard init done 下面我们输入了一个回车，所以显示出了个空行，这说明我们对回车键的处理是正确的。然后写入了小写字母 abcde，然后我按下了<caps lock>键，在屏幕上键入了 fghi，屏幕上输出了大写字母 FGHI，接着我又按住<L-shift>键入了 jklm，屏幕上输出了小写字母 jklm。回车换行，键入 i 键，屏幕输出了大写 I，随后我关闭<caps lock>键，继续键入后面的字符，其中的字符'!'是通过<R-shift>+1 键入的。另外，在以上键入过程中我多次按下<backspace>删除之前的字符，目前来说一切正常。

套用登月第一人阿姆斯特朗的那句名言：本节所做的工作对于计算机来说仅仅是一小步，但对于咱们来说却是一大步。

咱们暂时取得了阶段性的胜利，终于有和计算机交流的机会了，不过更多的交流还在后面，下节咱们继续努力。

10.5　环形输入缓冲区

到现在，咱们的键盘驱动仅能够输出咱们所键入的按键，这还没有什么实际用途。在键盘上操作是为了与系统进行交互，交互的过程一般是键入各种 shell 命令，然后 shell 解析并执行。

shell 命令是由多个字符组成的，并且要以回车键结束，因此咱们在键入命令的过程中，必须要找个缓冲区把已键入的信息存起来，当凑成完整的命令名时再一并由其他模块处理。

本节咱们要构建这个缓冲区。

10.5.1　生产者与消费者问题简述

在构建缓冲区之前，咱们得知道它的设计思路，打算解决哪些问题，因此还是要先来点“前奏”。

我们知道，在计算机中可以并行多个线程，当它们之间相互合作时，必然会存在共享资源的问题，这是通过“线程同步”来解决的。

“线程同步”是操作系统课程中必讲的内容，而诠释“线程同步”最典型的例子就是著名的“生产者与消费者问题”，初次接触它的时候真的很头疼。

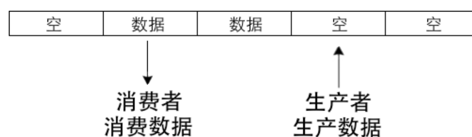
“同步”是指多个线程相互协作，共同完成一个任务，属于线程间工作步调的相互制约。“互斥”是指多个线程“分时”访问共享资源。

生产者与消费者问题是描述多个线程协同工作的模型，当初是由荷兰人 Dijkstra 为演示信号量而提出的，信号量解决了协同工作中的“同步”和“互斥”。

生产者与消费者模型如图 10-23 所示。

下面结合图 10-23 和大伙儿复习下这个经典模型。

有一个或多个生产者、一个或多个消费者和一个固定大小的缓冲区，所有生产者和消费者共享这同一个缓冲区。生产者生产某种类型的数据，每次放一个到缓冲区中，消费者消费这种数据，每次从缓冲区中消费一个。同一时刻，缓冲区只能被一个生产者或消费者使用。当缓冲区已满时，生产者不能继续往缓冲区中添加数据，当缓冲区为空时，消费者不能在缓冲区中消费数据。



▲图 10-23　生产者与消费者

概念介绍完了，解释得似乎还是很抽象，不过好在很多计算机中的设计思路都来自于生活，咱们还是拿生活中具体的例子再了解下。

一群学生的故事……每当上午快下课的时候，同学们都憋着一股劲，干吗？不是回寝室，更不是下课后请教老师问题，而是化身为追风少年冲向食堂。是啊……经过一上午繁重的脑力劳动再加上正处于长身体的时候，大伙儿早已经饥肠辘辘了，都想第一个冲到食堂开吃，不想在打饭时还要排队挨饿。

食堂运作流程一般都是这样的。

- 食堂里的大师傅负责烹饪。
- 将做好的饭菜放到打饭窗口里面的不锈钢菜盘里。窗口中所容纳菜盘的数量也是有限的，因此具有固定大小的菜品数。
- 随后学生在窗口排队打饭。

为方便陈述，将打饭窗口里面的不锈钢菜盆简称为窗口。

随着时间推移，打饭的学生越来越多，窗口里的菜越来越少，大师傅们又继续烹饪，及时往窗口中补充菜品。

现在咱们根据生产者与消费者的角色对号入座，大师傅是食物的生产者，打饭的窗口相当于食物的缓冲区，也是固定大小，学生相当于食物的消费者，您看生产者、缓冲区、消费者三样都齐了，这就是典型的生产者与消费者问题。

- 当窗口里少了某个菜品时，大师傅们就开始炒菜烹饪了，炒好了后就继续往窗口里放，直到窗口中的每样菜品都齐全为止，然后坐到一旁休息，如果刚炒完第一种菜时就已经有人排队打饭了，食堂工作人员就会喊一声：“可以打饭了”，这时候学生蜂拥而至……这就是生产者的行为，当缓冲区中数据不满时就生产，当缓冲区由空变不空时就唤醒消费者，只要缓冲区已满时就去休眠。
- 只要窗口中还有菜，不管菜品是否齐全，学生依然会在窗口继续打饭。直到窗口中一点菜都没有时学生打饭的活动才停止，然后朝大师傅大喊一声：“师傅，没菜啦”，随后淡定地排队，终于不再为过多的选择而纠结，这时候大师傅不再休息了，继续烹饪。这就是消费者的行为，当缓冲区中数据不空时就消费，只要缓冲区为空时就会休眠，休眠前会唤醒生产者。

总结一下，生产者与消费者问题描述的是：

对于有限大小的公共缓冲区，如何同步生产者与消费者的运行，以达到对共享缓冲区的互斥访问，并且保证生产者不会过度生产，消费者不会过度消费，缓冲区不会被破坏。

对于这种缓冲区的破坏，要么是对缓冲区访问溢出，也就是数据存取的地址超过了缓冲区的范围，要

么是缓冲区中的数据被破坏，也就是新数据把尚未读取的老数据覆盖。就像往篮子里放水果和取水果一样，篮子大小是固定的，能放多少也就能取多少，不能装得过多，否则水果就溢出来了，也不能取的过多，否则篮子底儿就被掏漏了，更不能因为篮子中无多余空间容纳新的水果，而把篮子中原来的水果压扁破坏（数据覆盖）占用旧水果的空间。

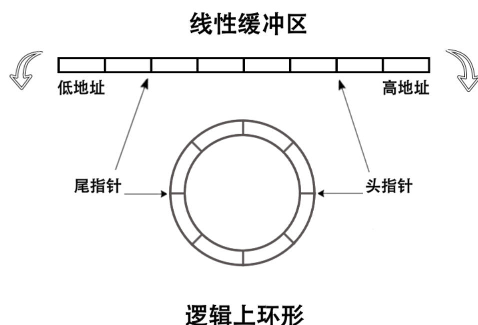
好啦，我相信生产者与消费者问题大伙儿至少在概念上已经清楚了，我们接下来要做的就是去用代码实现这种思想，咱们下节再叙。

10.5.2　环形缓冲区的实现

缓冲区是多个线程共同使用的共享内存，线程在并行访问它时难免会乱套，我们不能指望线程们会老老实实排好队，以串行的方式逐个使用缓冲区。缓冲区大小无关紧要，问题的关键在于缓冲区操作上，因此最好是在缓冲区的操作方法上下功夫，保证对缓冲区是互斥访问，并且不会对其过度使用，从而确保不会使缓冲区遭到破坏。也就是说，只要我们能够设计出合理的缓冲区操作方式，就能够解决生产者与消费者问题。

内存中的缓冲区就是用来暂存数据的一片内存区域，内存是按地址来访问的，因此内存缓冲区实际上是线性存储。但是我们可以设计出逻辑上非线性的内存缓冲区，通过合理的操作方式可以构造出任何我们想要的数据结构，在这里我要介绍下环形缓冲区。

环形缓冲区本质上依然是线性缓冲区，但其使用方式像环一样，没有固定的起始地址和终止地址，环内任何地址都可以作为起始和结束，如图 10-24 所示。



▲图 10-24　环形缓冲区

对于缓冲区的访问，我们提供两个指针，一个是头指针，用于往缓冲区中写数据，另一个是尾指针，用于从缓冲区中读数据。每次通过头指针往缓冲区中写入一个数据后，使头指针加 1 指向缓冲区中下一个可写入数据的地址，每次通过尾指针从缓冲区中读取一个数据后，使尾指针加 1 指向缓冲区中下一个可读入数据的地址，也就是说，缓冲区相当于一个队列，数据在队列头被写入，在队尾处被读出。

用线性空间实现这种逻辑上的环形空间，只要我们控制好头指针和尾指针的位置就好了，无论它们怎样变化，始终让它们落在缓冲区空间之内，当指针到达缓冲区的上边界后，想办法将指针置为缓冲区的下边界（通常是对缓冲区大小取模），从而使头尾指针形成回路，逻辑上实现环形缓冲区。这两个指针相当于缓冲区的游标，在缓冲区空间内来回滑动。

我们的环形缓冲区是个线性队列，队列可以用线性数据结构来实现，比如数组和链表，为了简单，咱们用数组来定义队列，实现环形缓冲区。

我们的环形缓冲区及其方法定义在 ioqueue.h 和 ioqueue.c 文件中，统一放到了 device 目录下，好啦，上菜，见代码 10-14。

代码 10-14　（project/c10/e/device/ioqueue.h）

```
 1 #ifndef __DEVICE_IOQUEUE_H
 2 #define __DEVICE_IOQUEUE_H
 3 #include "stdint.h"
 4 #include "thread.h"
 5 #include "sync.h"
 6
 7 #define bufsize 64
 8
 9 /* 环形队列 */
10 struct ioqueue {
11 // 生产者消费者问题
12     struct lock lock;
```

```
13  /* 生产者，缓冲区不满时就继续往里面放数据，
14   * 否则就睡眠，此项记录哪个生产者在此缓冲区上睡眠 */
15    struct task_struct* producer;
16
17  /* 消费者，缓冲区不空时就继续从里面拿数据，
18   * 否则就睡眠，此项记录哪个消费者在此缓冲区上睡眠 */
19    struct task_struct* consumer;
20    char buf[bufsize];                  // 缓冲区大小
21    int32_t head;                       // 队首，数据往队首处写入
22    int32_t tail;                       // 队尾，数据从队尾处读出
23 };
…略
```

struct ioqueue 结构便是咱们定义的环形缓冲区，它包括六个成员，其中：

lock 是本缓冲区的锁，每次对缓冲区操作时都要先申请这个锁，从而保证缓冲区操作互斥。

producer 是生产者，此项来记录当缓冲区满时，在此缓冲区睡眠的生产者线程。

consumer 是消费者，此项来记录当缓冲区空时，在此缓冲区睡眠的消费者线程。

buf[bufsize]是定义的缓冲区数组，其大小为 bufsize，在上面用 define 定义为 64。

head 是缓冲区队列的队首地址，tail 是队尾地址。

下面看对应的方法，见代码 10-15。

代码 10-15 （project/c10/e/device/ioqueue.c）

```
 1 #include "ioqueue.h"
 2 #include "interrupt.h"
 3 #include "global.h"
 4 #include "debug.h"
 5
 6 /* 初始化 io 队列 ioq */
 7 void ioqueue_init(struct ioqueue* ioq) {
 8    lock_init(&ioq->lock);     // 初始化 io 队列的锁
 9    ioq->producer = ioq->consumer = NULL;  // 生产者和消费者置空
10    ioq->head = ioq->tail = 0; //队列的首尾指针指向缓冲区数组第 0 个位置
11 }
12
13 /* 返回 pos 在缓冲区中的下一个位置值 */
14 static int32_t next_pos(int32_t pos) {
15      return (pos + 1) % bufsize;
16 }
17
18 /* 判断队列是否已满 */
19 bool ioq_full(struct ioqueue* ioq) {
20    ASSERT(intr_get_status() == INTR_OFF);
21    return next_pos(ioq->head) == ioq->tail;
22 }
23
24 /* 判断队列是否已空 */
25 static bool ioq_empty(struct ioqueue* ioq) {
26    ASSERT(intr_get_status() == INTR_OFF);
27    return ioq->head == ioq->tail;
28 }
29
30 /* 使当前生产者或消费者在此缓冲区上等待 */
31 static void ioq_wait(struct task_struct** waiter) {
32    ASSERT(*waiter == NULL && waiter != NULL);
33    *waiter = running_thread();
34    thread_block(TASK_BLOCKED);
35 }
36
37 /* 唤醒 waiter */
38 static void wakeup(struct task_struct** waiter) {
39    ASSERT(*waiter != NULL);
40     thread_unblock(*waiter);
41    *waiter = NULL;
42 }
43
44 /* 消费者从 ioq 队列中获取一个字符 */
45 char ioq_getchar(struct ioqueue* ioq) {
```

```
46     ASSERT(intr_get_status() == INTR_OFF);
47
48 /* 若缓冲区（队列）为空，把消费者 ioq->consumer 记为当前线程自己，
49  * 目的是将来生产者往缓冲区里装商品后，生产者知道唤醒哪个消费者，
50  * 也就是唤醒当前线程自己*/
51    while (ioq_empty(ioq)) {
52        lock_acquire(&ioq->lock);
53        ioq_wait(&ioq->consumer);
54        lock_release(&ioq->lock);
55    }
56
57    char byte = ioq->buf[ioq->tail];     // 从缓冲区中取出
58    ioq->tail = next_pos(ioq->tail);     // 把读游标移到下一位置
59
60    if (ioq->producer != NULL) {
61         wakeup(&ioq->producer);         // 唤醒生产者
62    }
63
64    return byte;
65 }
66
67 /* 生产者往 ioq 队列中写入一个字符 byte */
68 void ioq_putchar(struct ioqueue* ioq, char byte) {
69     ASSERT(intr_get_status() == INTR_OFF);
70
71 /* 若缓冲区（队列）已经满了，把生产者 ioq->producer 记为自己，
72  * 为的是当缓冲区里的东西被消费者取完后让消费者知道唤醒哪个生产者，
73  * 也就是唤醒当前线程自己*/
74     while (ioq_full(ioq)) {
75        lock_acquire(&ioq->lock);
76        ioq_wait(&ioq->producer);
77        lock_release(&ioq->lock);
78     }
79     ioq->buf[ioq->head] = byte;         // 把字节放入缓冲区中
80     ioq->head = next_pos(ioq->head);    // 把写游标移到下一位置
81
82     if (ioq->consumer != NULL) {
83        wakeup(&ioq->consumer);          // 唤醒消费者
84     }
85 }
86
```

咱们从上往下说，ioqueue_init 函数接受一个缓冲区参数 ioq，用于初始化缓冲区 ioq。此函数负责三样工作，先通过初始化 io 队列的锁，再将生产者和消费者置为 NULL，最后再将缓冲区的队头和队尾置为下标 0。

next_pos 函数接受一个参数 pos，功能是返回 pos 在缓冲区中的下一个位置值，它是将 pos+1 后再对 bufsize 求模得到的，这保证了缓冲区指针回绕着数组 buf，从而实现了环形缓冲区。

ioq_full 函数接受一个缓冲区参数 ioq。功能是返回队列是否已满，若已满则返回 true，否则返回 false。原理是“next_pos(ioq->head) == ioq->tail”，判断下一个队首位置是否和队尾位置相等，即是否会碰撞。从这一点看出，虽然缓冲区大小 bufsize 是 64 字节，但其最大容量为 63 字节。

ioq_empty 函数接受一个缓冲区参数 ioq。功能是返回队列是否为空，若空则返回 true。原理是判断 ioq->head 是否等于 ioq->tail，若头尾相等则为空。

ioq_wait 函数接受一个参数 waiter，它是 pcb 类型的二级指针，因此传给它的实参将是线程指针的地址，函数功能是使当前线程睡眠，并在缓冲区中等待。估计大伙儿都猜到了，传给 waiter 的实参一定是缓冲区中的成员 producer 或 consumer。在函数体内就做了两件事，将当前线程记录在 waiter 指向的指针中，也就是缓冲区中的 producer 或 consumer，因此*waiter 相当于 ioq->consumer 或 ioq->producer。随后调用“thread_block(TASK_BLOCKED)”将当前线程阻塞。

wakeup 函数接受一个参数 waiter，它同样也是 pcb 类型的二级指针，因此传给它的实参也是缓冲区中的成员 producer 或 consumer。函数功能就是通过“thread_unblock(*waiter)”唤醒*waiter（生产者或消费者），随后将*waiter 置空。

咱们对缓冲区操作的数据单位大小是 1 字节，即生产者每次往缓冲区中放一字节数据，消费者每次从

缓冲区中取一字节数据。

ioq_getchar 函数接受一个缓冲区参数 ioq，函数功能是从 ioq 的队尾处返回一个字节，这属于从缓冲区中取数据，因此 ioq_getchar 是由消费者线程调用的。

函数体中，先通过“while(ioq_empty(ioq))”循环判断缓冲区 ioq 是否为空，如果为空就表示没有数据可取，只好先在此缓冲区上睡眠，直到有生产者将数据添加到此缓冲区后再被叫醒重新取数据。但是你懂的，消费者有可能有多个，它们之间是竞争的关系，醒来后有可能别的消费者刚刚把缓冲区中的数据取走了，因此在当前消费者被叫醒后还要再判断缓冲区是否为空才比较保险，所以用 while 循环来重复判断。

while 循环体中先通过“lock_acquire(&ioq->lock)”申请缓冲区的锁，持有锁后，通过“ioq_ wait(&ioq->consumer)”将自己阻塞，也就是在此缓冲区上休眠。您看，这里传给 ioq_wait 的实参就是缓冲区的消费者&ioq->consumer，此项用来记录哪个消费者没有拿到数据而休眠。这样等将来某个生产者往缓冲区中添加数据的时候就知道叫醒它继续拿数据了。醒来后执行“lock_release(&ioq->lock)”释放锁。

在 while 循环判断中，如果缓冲区不为空的话，通过代码“byte = ioq->buf[ioq->tail]”从缓冲区队尾获取 1 字节的数据，接着通过“ioq->tail = next_pos(ioq->tail)”将队尾更新为下一个位置。

在消费者读取了一个字节后，缓冲区就腾出一个数据单位的空间了，这时候要判断一下是否有生产者在此缓冲区上休眠。若之前此缓冲区是满的，正好有生产者来添加数据，那个生产者一定会在此缓冲区上睡眠。因此要判断 ioq->producer 是否不等于 NULL，如果不等于 NULL，这说明之前有生产者线程在调用函数 ioq_putchar（生产者往添加缓冲区中添加数据的方法，后面马上介绍）往缓冲区中添加数据时因为缓冲区满而休眠了，既然现在缓冲区已被当前消费者线程腾出一个数据单位的空间了，此时应该叫醒生产者继续往缓冲区中添加数据。因此调用“wakeup(&ioq->producer)”唤醒生产者。之后通过 return byte 返回获取数据。

ioq_putchar 函数接受两个参数，一个是缓冲区参数 ioq，另一个是待加入字节数据 byte，函数功能是往缓冲区 ioq 中添加 byte，这是由生产者线程调用的。

在函数体中也是先通过 while 循环判断缓冲区 ioq 是否为满，如果满了的话，先申请缓冲区的锁 ioq->lock，然后通过调用“ioq_wait(&ioq->producer)”将自己阻塞并登记在缓冲区 ioq 的成员 producer 中，这样消费者便知道唤醒哪个生产者了。随后释放锁。

如果缓冲区不满的话，通过“ioq->buf[ioq->head] = byte”，将数据 byte 写入缓冲区的队首 ioq->head。随后通过“ioq->head = next_pos(ioq->head)”将队首更新为下一位置。

这里依然要判断是否有消费者在此缓冲区上休眠。若之前此缓冲区为空，恰好有消费者来取数据，因此会导致消费者休眠。现在当前生产者线程已经往缓冲区中添加了数据，现在可以将消费者唤醒让它继续取数据了。如果 ioq->consumer 不等于 NULL，这说明之前已经有消费者线程因为缓冲区空而休眠，被登记在 ioq->consumer，因此调用“wakeup(&ioq->consumer)”将消费者唤醒。

有关环境缓冲区的实现到这就介绍完了，我知道您已经迫不及待想实际测试了，好啦，咱们赶紧进入下一节去应用它。

10.5.3　添加键盘输入缓冲区

虽然我们的环形缓冲区支持多个生产者和消费者，但目前我们应用的场合非常简单，只是用在单一生产者和单一消费者的环境中，即生产者是键盘驱动，消费者是将来的 shell，那现在您知道了，本节要将在键盘驱动中处理的字符存入环形缓冲区当中。

本节改动比较小，没啥可说的啦，直接上代码。请大伙儿参见代码 10-16。

代码 10-16　（project/c10/e/device/keyboard.c）

```
…略
38 struct ioqueue kbd_buf;    // 定义键盘缓冲区
…略
193      if (cur_char) {
…略
205   /* 若 kbd_buf 中未满并且待加入的 cur_char 不为 0,
206    * 则将其加入到缓冲区 kbd_buf 中 */
```

```
207         if (!ioq_full(&kbd_buf)) {
208             put_char(cur_char);     // 临时的
209             ioq_putchar(&kbd_buf, cur_char);
210         }
211         return;
212       }
…略
231 /* 键盘初始化 */
232 void keyboard_init() {
233    put_str("keyboard init start\n");
234    ioqueue_init(&kbd_buf);
235    register_handler(0x21, intr_keyboard_handler);
236    put_str("keyboard init done\n");
237 }
238
```

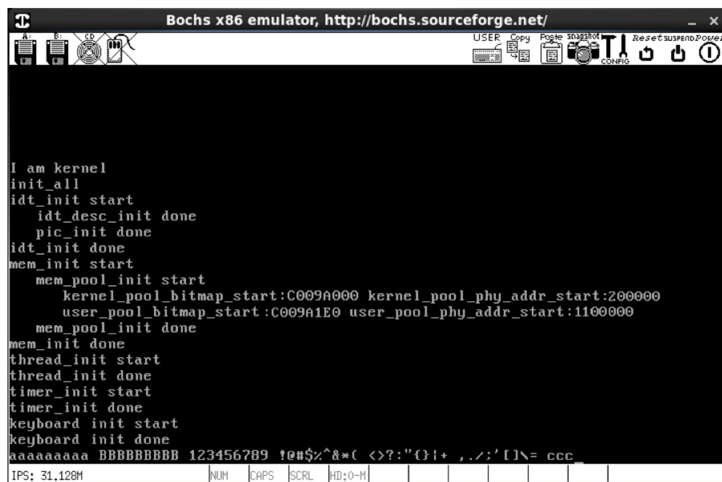
代码 10-16 中，第 38 行定义了 kbd_buf，这就是之前所介绍的环形缓冲区，我们用它来做键盘的缓冲区。

为了能够使用 kbd_buf，在 keyboard_init 函数中添加了对其初始化的调用——“ioqueue_init (&kbd_buf)”。

应用缓冲区的地方是在键盘驱动中啦，在将扫描码转换为字符之后，在第 207 行，先通过“if (!ioq_full(&kbd_buf))”判断缓冲区是否已满，如果未满则通过“ioq_putchar(&kbd_buf, cur_char)”将转换后的字符 cur_char 加入到缓冲区中。

不过在此之前，我们还通过“put_char(cur_char)”来打印，这是临时放在这的，为的是演示缓冲区写满的情况。理论情况是咱们缓冲区只支持 63 个字节，多输入的字符将不再响应，一会儿咱们要验证这个功能。

编译之后在 bochs 中运行，大伙儿先看效果吧，如图 10-25 所示。



▲图 10-25　写满缓冲区

大伙儿注意，图 10-25 最下面一行是我键入的字符串，字符串分为 7 组，前 6 组中，每组第 10 个字母为空格。第 7 组是字符串 ccc，因此一共是 63 个字符。这是咱们缓冲区的最大容量了，之后我再怎么键入键盘屏幕都不会再输出字符。这对应于第 207 行的实际代码“if (!ioq_full(&kbd_buf))”，缓冲区满了，条件不成立，因此执行后面的 return 返回。

仅仅这样测试似乎还意犹未尽，咱们目前只有生产者，还没有测试消费者呢，下一节，咱们来个自给自足，由键盘中断往缓冲区里生产数据，再创建一个新线程从缓冲区里消费数据。

10.5.4　生产者与消费者实例测试

本节中的代码纯粹为演示生产者与消费者，和咱们的实际项目没关系，算是内核开发中的小插曲。

在咱们现实的应用中也不是键盘上所有操作都被处理，当键入的按键数量超过缓冲区大小时，有可能主机会发出“嘀嘀”地提示音以警告按键操作过多。咱们的处理也类似，当缓冲区满了的时候，键盘中断程序不会继续往此缓冲区上添加数据，因此后来的键入操作会被丢弃。

目前只有生产者，也就是键盘驱动，现在咱们添加两个消费者线程 k_thread_a 和 k_thread_b，让它们从键盘缓冲区中消费数据。

为了测试键盘驱动，咱们目前只打开了键盘中断，为了让消费者线程能够上处理器运行，现在必须要打开时钟中断。修改下 interrupt.c，将时钟中断和键盘中断都打开，请见代码 10-17。

代码 10-17　（project/c10/e_PandC/kernel/interrupt.c）

```
40 static void pic_init(void) {
…略
54 /* 测试键盘，只打开键盘中断，其他全部关闭 */
55    outb (PIC_M_DATA, 0xfc);
56    outb (PIC_S_DATA, 0xff);
…略
59 }
```

第 55 行，通过"outb (PIC_M_DATA, 0xfc)"，写入数值 0xfc 到主片的中断屏蔽寄存器，0xfc 低 2 位为 0，这两位对应的是时钟中断和键盘中断，它们为 0，表示不屏蔽中断，即打开这两个中断。interrupt.c 的修改到此为止。

键盘缓冲区是全局数据结构，它在生产者和消费者之间共享，因此在添加生产者线程之前，还要在 keyboard.h 中添加此缓冲区的声明，这样外部函数就可以访问到它，见代码 10-18。

代码 10-18　（project/c10/e_PandC/device/keyboard.h）

```
 1 #ifndef __DEVICE_KEYBOARD_H
 2 #define __DEVICE_KEYBOARD_H
 3 void keyboard_init(void);
 4 extern struct ioqueue kbd_buf;
 5 #endif
```

第 4 行添加了 kbd_buf 的外部声明，在消费者线程中会用到它。

下面我们在 main.c 中添加消费者线程，请见代码 10-19。

代码 10-19　（project/c10/e_PandC/kernel/main.c）

```
…略
 7 /* 临时为测试添加 */
 8 #include "ioqueue.h"
 9 #include "keyboard.h"
10
11 void k_thread_a(void*);
12 void k_thread_b(void*);
13
14 int main(void) {
15     put_str("I am kernel\n");
16     init_all();
17     thread_start("consumer_a", 31, k_thread_a, " A_");
18     thread_start("consumer_b", 31, k_thread_b, " B_");
19     intr_enable();
20     while(1);
21     return 0;
22 }
23
24 /* 在线程中运行的函数 */
25 void k_thread_a(void* arg) {
26     while(1) {
27        enum intr_status old_status = intr_disable();
28        if (!ioq_empty(&kbd_buf)) {
29            console_put_str(arg);
30            char byte = ioq_getchar(&kbd_buf);
31            console_put_char(byte);
32        }
33        intr_set_status(old_status);
34     }
35 }
36
37 /* 在线程中运行的函数 */
38 void k_thread_b(void* arg) {
```

```
39     while(1) {
40        enum intr_status old_status = intr_disable();
41        if (!ioq_empty(&kbd_buf)) {
42            console_put_str(arg);
43            char byte = ioq_getchar(&kbd_buf);
44            console_put_char(byte);
45        }
46        intr_set_status(old_status);
47     }
48 }
```

这里咱们只添加了两个消费者线程：k_thread_a 和 k_thread_b，这两个线程是咱们的老朋友了，这次对它们稍加修改，让它们从键盘缓冲区中获取数据，成为消费者线程。主线程只用了 while(1)来悬停，并没有作为消费者，这里不想贴太多冗余代码，留待大伙儿自行测试吧（话说我这里悄悄测试过，结果正常）。

k_thread_a 和 k_thread_b 是一样的，拿 k_thread_a 举例，在 while 循环体中不断输出参数 arg 和从键盘缓冲区中获取的数据。不过在输出它们之前，先通过 intr_disable 关中断，其实这不是必须的，仅仅是“迁就”函数 ioq_getchar 和 ioq_putchar 中的“ASSERT(intr_get_status() == INTR_OFF)”，咱们对缓冲区的操作必须在关中断的情况下进行。

第 17～18 行也有了修改，通过 thread_start 将线程 k_thread_a 封装为线程，并命名为 consumer_a，将线程 k_thread_b 命名为 consumer_b。为了区别这两个消费者，咱们传给它们的参数分别为“ A_”和“ B_”（注意字母前有个空格），让它们在从缓冲区中取数据前先各自输出自己的参数，这样容易分清是谁从缓冲区中拿走了数据。

编译运行之后，咱们看看执行效果，如图 10-26 所示。

```
mem_init done
thread_init start
thread_init done
timer_init start
timer_init done
keyboard init start
keyboard init done
 B_k A_k B_k B_k B_k B_k B_k B_k A_k A_k A_k A_k B_k A_k A_k A_k A_k A_k A_k A_k
 A_k A_k A_k A_k A_k A_k A_k A_k A_k A_k B_k A_k A_k A_k A_k B_k B_k B_k B_k B_k
 B_k B_k B_k B_k B_k B_k A_k A_k A_k A_k B_k A_k A_k A_k B_k A_k A_k A_k A_k B_k
 A_k A_k A_k A_k A_k A_k B_k A_k A_k A_k A_k B_k B_k A_k B_k B_k B_k_
IPS: 21.711M          NUM  CAPS  SCRL  HD:0-M
```

▲图 10-26　生产者与消费者实例

在 bochs 中执行 C 命令持续执行后，程序输出“keyboard init done”后悬停。这时候我在键盘上按住 k 键不松手，屏幕上一直交替输出“B_k”，和“A_k”，这符合咱们的预期，说明 ioqueue 起作用了。

好啦，咱们又往前走了一大步，本章到此结束，下章再见。

第 11 章　用户进程

一直以来我们的程序都在最高特权级 0 级下工作，这意味着任何程序都和操作系统平起平坐，可以改动任何系统资源。如果不改变这种现状的话，某个不听话的程序甚至可以给操作系统致命一击，取而代之……后果将不可估量。

操作系统存在的目的之一就是资源管理，这里面就包括安全隔离的内容。要想服众，必须得拥有至高无上的权力，因此，被管理的程序不能具有太高的特权，必须要比操作系统的权力低，通常这个被管理的对象就是用户进程。我们把用户程序的特权级降级到 3 级，这样操作系统才能高枕无忧。

本章将开始介绍用户进程及相关的内容。

11.1 为什么要有任务状态段 TSS

早在第 5 章中咱们已经简单介绍过 TSS 了，当时是为了介绍和 I/O 端口特权相关的 I/O 位图，此位图位于 TSS 中。现在咱们接着把和 TSS 有关的剩下部分说完。

在操作系统课程中，老师经常说：“Linux 任务切换未采用 Intel 的做法，而是用了一套自己的方法，只是用了 TSS 的一小部分功能”。对于这句话，如果最初就不明白的话，继续深问下去似乎也帮助不大，甚至会击垮本来已经建立的知识体系，让人更加糊涂。

其实咱们在学习过程中会遇到很多类似的情况：如果一开始就不懂，之后无论怎么解释都不明白。造成这一现象可能的原因是。

（1）自己在这方面基础薄弱，该了解的基础知识不足。

（2）或许自己只是某个知识点没搞清楚，从而导致整个知识链是混乱的，但自己并不知道是哪个知识点理解错了。

（3）对方站在他自己的知识层面上解释，因此对咱们来说显得太 high-level，不够通透。

（4）对方自己就不是特别明白，只是对此知识熟练了，因此认同、接受了而已。

因此，我在这里和大伙儿说说 TSS 和任务切换的渊源吧，希望能够帮助当初和我一样对此感到困惑的兄弟们。

11.1.1　多任务的起源，很久很久以前……

我们平时工作中所用的文档都是用某种应用程序创建的，应用程序一般是由某种编译器按照某种语言规则编译出来的，编译器也是程序，它为了编译出某操作系统平台上的应用程序，必须与宿命操作系统紧密耦合，因此编译器依赖宿主系统，“相当于”其应用程序，咱捋捋这个关系：文档是应用程序的应用，应用程序是编译器的应用，编译器又是操作系统的应用。那操作系统是谁的应用？当然是硬件的应用啦。

任何软件的运行都需要落实到硬件上，操作系统权力再高也只是软件，它的运行同样必须由硬件来支撑，只不过它和一般软件的区别是：操作系统是直接和硬件打交道的软件，普通的应用软件可没这个待遇。

操作系统能做什么，完全取决于硬件给它提供的功能支持。这就像驾驶汽车一样，如何驱动这辆车，要看车本身提供了哪些驱动的方法，比如有方向盘、油门等，而驾驶员也只是灵活应用这些驱动方法而已。我们的操作系统也只是众多计算机硬件的驾驶员。

操作系统最直接控制的就是 CPU，要想让 CPU 这颗奔腾的心永远地跳下去，首先必须把内存分成段，把内存按“内存块”访问，其次必须让代码段寄存器 CS 和指令寄存器[E]IP 指向下一条待执行的指令。这是人家 CPU 开发厂商规定的，也就是说，要想让这颗 CPU 帮咱们执行指令，咱们就必须遵守人家厂商的规矩，否则干脆就不要用人家的 CPU。因此，不管 CS 和[E]IP 指向的到底是什么，人家 CPU 就是把 CS:[E]IP 指向的内容当成指令，咱们要是任性地把它们指向普通的数据，吃苦头的可是咱们自己。这些规矩看上去有些冷酷无情，但又合理得让人无言以对。

现在该说说 TSS 的事了。TSS 是 Task State Segment 的缩写，即任务状态段，早在第 5 章介绍 I/O 特权级的时候已经部分地介绍过它了，忘了的话，大伙儿可以回头看看第 5 章中的 TSS 简介，其结构已经在图 5-46 展示了。迄今为止我们只介绍了 TSS 中的 I/O 位图，它还有很多其他字段没介绍呢，除了和 I/O 端口特权相关外，它到底还用来做什么呢？

还是容我慢慢和您介绍吧，话得从头说起。

我们知道，软件的能力取决于硬件的支持，但操作系统和 CPU 之间的能力约束并不是绝对单向的，说白了并不是说谁单方向决定了对方的能力，原因是：虽说厂商才是功能提供者，但需求方才是促进硬件发展的原因和动力。咱拿骑自行车举例，骑车的时候，要想让自行车停下来，咱们确实可以用脚刹车，抛开安全性不说，这样太费鞋了。因此自行车制造商便提供了手闸，想停车的时候双手一捏闸就行了，不过咱们依然可以用“脚刹”来代替。操作系统也是一样，它想实现某种功能，如果软件上的解决方式不好，或者干脆解决不了，就只能向 CPU 等硬件厂商提需求，让硬件一级直接支持，硬件厂商因此给硬件增加新的功能，从而使硬件得到了发展，软件（操作系统）便可以利用硬件的新功能，因此也变得更加强大。之后硬件再去支持新的需求，如此良性循环下去，硬件和软件逐渐形成规模，这就是相互促进发展的结果。好啦，不能扯太远了，咱们回来说 TSS。

大伙儿知道，起初的 CPU 只支持单任务，但后来随着多任务的需求越来越迫切，操作系统厂商和 CPU 厂商便开始构想多任务的方案了。不过，这次可不像让自行车停下来那样简单了，对于自行车我们还可以将就着用“脚刹”来实现停车的功能，但这次面临的困难相当于开汽车，汽车的动能太大了，驾驶员再用脚刹也是以卵击石，因此，最好的办法是汽车本身提供个制动方法，也就是刹车。

操作系统毕竟只是软件，能做的事实在有限，在软件层面实现的多任务调度有点类似今天的用户态多线程，效率不高且安全性上有诸多问题，于是向 CPU 厂商提了这个需求，希望硬件给予多任务的原生支持。

硬件厂商为此提供了硬件解决方案，其中最主要的就是 LDT 和 TSS，我们下节再说。

11.1.2　LDT 简介

友情提示，本节只是作为内容扩展，为的是满足部分读者的好奇心，咱们在项目中并不会用到 LDT，所以大伙儿可以选择不看。不过话说回来了，正是由于好奇心特别大，所以才会想了解操作系统的原理，既然选择了本书，岂有不看的道理。

大伙儿知道，程序是一堆数据和指令的集合，它们只有被加载到内存并让 CPU 的寄存器中指向它们后，CPU 才能执行该程序。程序从文件系统上被加载到内存后，位于内存中的程序便称为映像，也称为任务。

在 IA32 架构的 CPU 上，内存被设计成需要按照分段的方式来访问。软件只是硬件的应用，因此对于咱们软件工程师来说，咱们要在这种 CPU 上开发程序，内存分段访问这是硬性规定，必须遵守。尽管在通常情况下，为了使程序组织清晰有条理，程序员都会将数据分类存储：数据集中连续地放一起，指令集中连续地放一起。但这也只是出于审美，并不是强制要求。咱们只要按照 CPU 的规定，让代码段寄存器 CS:[E]IP 指向程序映像中的指令就行了，让数据段寄存器 DS 指向映像中的数据就行了，如果不这么做，CPU 就会抛异常。比如把 CS 指向了映像中的数据部分，CPU 不会检查待执行的指令是否合法，因为它检查不了，原因是 CPU 中的指令繁多，万一某个原本作为数据的二进制串“恰好”等于某个指令呢。这里虽然用了“恰好”，但其实这并不是少数，而是大部分情况下都能够被 CPU 的指令部件识别成某种指令，只不过会错误地执行下去，直到无法执行为止。同样一组数在不同的上下文中不同的意义，就像生日本身是个数字，但在密码输入框中即使输入的是生日数字也只会被当成密码。在计算机中也一样，同样一组二进制数在不同的上下文中

也有不同的意义，具体代表什么，取决于 CPU 怎么看待它。CPU 只把 CS:[E]IP 指向的内存当成指令，把 DS 指向的内存当作普通数据，因此必须人为地保证填充到这些段寄存器中的值是正确的。咱们只要往对应的寄存器中写入合适的值就成了，其他的咱们不用管，由处理器内部的处理框架自动完成。这就像咱们在软件开发过程中用到的框架一样，只不过这次的框架是由硬件 CPU 提供的。

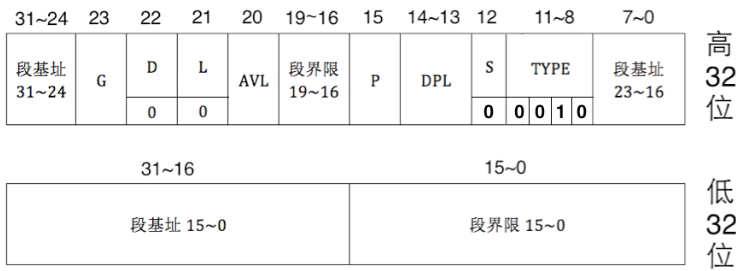
按照内存分段的方式，内存中的程序映像自然被分成了代码段、数据段等资源，这些资源属于程序私有的部分，因此 Intel 建议，为每个程序单独赋予一个结构来存储其私有的资源，这个结构就是 LDT。

LDT 是 Local Descriptor Table 的缩写，即局部描述符表。说到这您肯定猜到了，LDT 和 GDT 是对应的，GDT 是全局描述符表，里面存放的是用于全局的内存描述符。描述符的功能就是描述一段内存区域的作用及属性，它只是对应内存区域的身份证。

LDT 属于任务私有的结构，它是每个任务都有的，其位置自然就不固定。为了使用它，最起码得先能够找到它。我想您肯定想到了，GDT 是全局唯一的结构，它的位置是固定且已知的，它已经由 LGDT 指令将其起始地址及偏移量存储到 GDTR 寄存器中。因此，LDT 必须像其他描述符那样在 GDT 注册，之后便能够用选择子找到它。

描述符的作用是描述一段内存区域的属性，其中最重要的属性是内存区域的起始地址及偏移大小。描述符表是位于内存中的表格，因此，描述符表依然可以用描述符来表示。

在这之前，咱们都是把描述符写入描述符表，如段描述符写入 GDT 中，从来没有把描述符表再写入其他表的情况。LDT 虽然是描述符“表”，为了在 GDT 中注册，必须也得为它找个描述符，用此描述符来描述某任务的 LDT 的起始地址及偏移大小，此描述符便称为 LDT 描述符，其结构如图 11-1 所示。



▲图 11-1　LDT 描述符格式

在 LDT 中，描述符的 D 位和 L 位固定为 0。

LDT 描述符属于系统段描述符，因此 S 为 0。在 S 为 0 的前提下，若 TYPE 的值为 0010，这表示此描述符是 LDT 描述符。其他字段意义同段描述符相同，不再赘述。

现在能够找到 LDT 了，但是如何使用它呢？

CPU 使用某个表，肯定不只是找到一个描述符就行了，描述符的目的是为了告诉 CPU 描述符所对应区域的起始地址及偏移大小。想想 CPU 是如何找到 GDT 的，在寄存器 GDTR 中的内容便是 GDT 的起始地址及偏移量大小，只要 GDT 被 Lgdt 指令加载后，CPU 在 GDTR 中便能找到 GDT。

和 GDT 一样，CPU 专门准备了个寄存器来存储其位置及偏移量，想必您又猜到了，对，这就是 LDTR。CPU 同样也准备了配套的指令，就是 lldt，用此指令能够将 ldt 加载到 LDTR 寄存器。lldt 的指令格式为：

lldt“16 位通用寄存器”或“16 位内存单元”

不管操作数中寄存器还是内存，其值必须是 LDT 选择子。

对比一下，加载 GDT 的指令是 lgdt，其格式是：

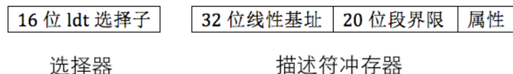
lgdt“16 位内存单元”&“32 位内存单元”

前 16 位表示 GDT 的偏移大小，后 32 位表示 GDT 的起始地址。区别是，lgdt 的操作数是 GDT 表的偏移量及起始地址，而 lldt 的操作数是 ldt 在 GDT 中的选择子。这确实让人感觉很乱，毕竟 LDT 和 GDT 一

样都是描述符表，为了定位一个表，CPU 关心的是表的起始位置及偏移界限（大小减 1），按理说 lldt 的操作数也应该为“16 位内存单元（大小偏移量）”&“32 位内存单元（起始地址）”。但这么做是有原因的，这是 CPU 进行安全检查的方式，用选择子在 GDT 中索引 LDT 描述符，这样可以套用之前咱们介绍过的引用段描述符时的特权级检查。

LDTR 寄存器结构如图 11-2 所示。

LDTR 分为两个部分，选择器是中 16 位的 LDT 选择子，描述符缓冲器中是 LDT 的起始地址及偏移大小等属性。



▲图 11-2　LDTR 寄存器

LDT 中的描述符全部用于指向任务自己的内存段，该如何引用它们呢？

大伙儿还记得选择子的结构吗？选择子是 16 位的，其高 13 位是索引值，用来在 GDT 或 LDT 中索引段描述符，第 0～1 位 RPL，表示请求特权级，第 2 位是 TI 位，此位用来指定选择子中的高 13 位是在 GDT 中索引段描述符，还是在 LDT 中索引段描述符。TI 位也就是 Table Indicator，当此位为 1 时，表示从 LDT 中检索，反之当此位为 0 时，表示从 GDT 中检索选择子。

补充一点，当 TI 为 0 时，CPU 到 GDTR 中找 GDT，当 TI 为 1 时，CPU 到 LDTR 中找 LDT。

与 GDT 不同的是 LDT 中的第 0 个段描述符是可用的。想想看，为什么 GDT 中第 0 个段描述符不可用？这么做的原因很简单，是担心选择子未初始化，这种未初始化的选择子其值为 0，因此会选择到 GDT 中的第 0 个段描述符，若选择到 GDT 中的第 0 个段描述符，CPU 便知道这是选择子未初始化造成的，于是通过抛异常来发现这种错误。要是从 LDT 中选择段描述符时，选择子的 TI 位必须为 1，这就确保了只有经过显式地初始化后才能从 LDT 中检索描述符，不存在忘记初始化的情况，因此 LDT 中的第 0 个段描述符是可用的。

选择子的高 13 位表示可索引的描述符范围，2 的 13 次方等于 8192，也就是说一个任务最多可定义 8192 个内存段。由于 LDT 描述符放在 GDT 中，如果任务是用 LDT 来实现的话，最多可同时创建 8192 个任务。

当前运行的任务，其 LDT 位于 LDTR 指向的地址，这样 CPU 才能从中拿到任务运行所需要的资源（指令和数据）。因此，每切换一个任务时，需要用 lldt 指令重新加载新任务的 LDT 到 LDTR。

虽然介绍了 LDT，但咱们并不打算使用它，因为每加入一个任务都需要在 GDT 中添加新的 LDT 描述符，还要重新加载 LDTR，比较麻烦。而我们是在平坦模型下编程，因此任务的实现已经用不着这么麻烦了。

有关 LDT 的部分就点到为止，咱们下节继续说 TSS。

11.1.3　TSS 的作用

单核 CPU 要想实现多任务，唯一的方案就是多个任务共享同一个 CPU，也就是只能让 CPU 在多个任务间轮转，让所有任务轮流使用 CPU，除此之外还真别无他法。

大家已经知道了 LDT 中是任务自己的私有资源，每个任务都有自己的 LDT，因此我们不需要担心多任务时，程序的运行资源会混乱。但，这还不够。

CPU 执行任务时，需要把任务运行所需要的数据加载到寄存器、栈和内存中，因为 CPU 只能直接处理这些资源中的数据，这是 CPU 在设计之初时工程师们决定的，属于“基因”里的内容，因此，任务（软件）在此类 CPU 上执行时，必须遵守此规定。于是问题来了，任务的数据和指令是 CPU 的处理对象，任务的执行要占用一套存储资源，如寄存器和内存，这些存储资源中装的是任务的数据和指令，它们属于 CPU 的大餐，但 CPU 很不情愿直接用内存这个低速的、不方便的容器就餐，它最喜欢的容器是寄存器，因为它的速度和 CPU 很般配，这才能让 CPU 吃得更爽。因此内存中的数据往往被加载到高速的寄存器后再处理，处理完成后，再将结果回写到低速的内存中，所以，任何时候，寄存器中的内容才是任务的最新状态。采取轮流使用 CPU 的方式运行多任务，当前任务在被换下 CPU 时，任务的最新状态，也就是寄存器中的内容应该找个地方保存起来，以便下次重新将此任务调度到 CPU 上时可以恢复此任务的最新状态，这样任务才能继续执行，否则就出错了。

的确是这个道理，CPU 厂商也是这么想的，Intel 的建议是给每个任务“关联”一个任务状态段，这就是 TSS（Task State Segment），用它来表示任务。

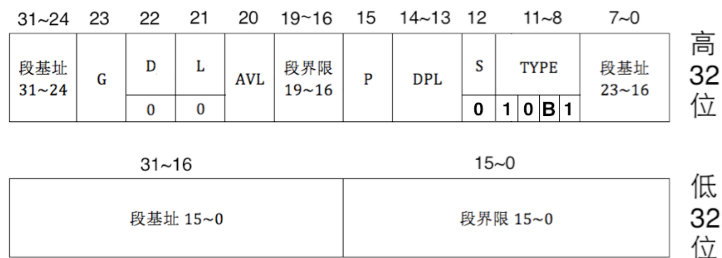
之所以称为“关联”，是因为 TSS 是由程序员“提供”的，由 CPU 来“维护”。“提供”就是指 TSS 是程

序员为任务单独定义的一个结构体变量，“维护”是指 CPU 自动用此结构体变量保存任务的状态（任务的上下文环境，寄存器组的值）和自动从此结构体变量中载入任务的状态。当加载新任务时，CPU 自动把当前任务（旧任务）的状态存入当前任务的 TSS，然后将新任务 TSS 中的数据载入到对应的寄存器中，这就实现了任务切换。TSS 就是任务的代表，CPU 用不同的 TSS 区分不同的任务，因此任务切换的本质就是 TSS 的换来换去。

CPU 如何知道 TSS 换了？在此先剧透一下：在 CPU 中有一个专门存储 TSS 信息的寄存器，这就是 TR 寄存器，它始终指向当前正在运行的任务，因此，“在 CPU 眼里”，任务切换的实质就是 TR 寄存器指向不同的 TSS，后面会介绍这方面内容。

注意，以上所说的“在 CPU 眼里”，是指 CPU 视角中任务的概念，CPU 原计划为每个任务关联一个 TSS，因此每个任务都必须有单独的 TSS，所以 TSS 就是任务的代表。而人类理解的任务切换，就是让 CPU 执行不同任务的代码段中的指令，说白了就是让 CPU 的 CS:[e]ip 指向不同任务的代码，即使是所有任务共享一个 TSS 也无所谓，这就是 Linux 的作法。好啦，这是后话，总之在 CPU 眼里任务切换就是 TSS 换来换去，而这只是 CPU 的美好愿景，Linux 并未这么做，我们也是。

TSS 和其他段一样，本质上是一片存储数据的内存区域，Intel 打算用这片内存区域保存任务的最新状态（也就是任务运行时占用的寄存器组等），因此它也像其他段那样，需要用某个描述符结构来“描述”它，这就是 TSS 描述符，TSS 描述符也要在 GDT 中注册，这样才能“找到它”。TSS 描述符结构如图 11-3 所示。



▲图 11-3　TSS 描述符格式

TSS 描述符属于系统段描述符，因此 S 为 0，在 S 为 0 的情况下，TYPE 的值为 10B1。我们这里关注一下 B 位，B 表示 busy 位，B 位为 0 时，表示任务不繁忙，B 位为 1 时，表示任务繁忙。

其他字段的意义与普通数据段类似，不再赘述，这里解释下什么是任务繁忙。

任务繁忙有两方面的含义，一方面就是指此任务是否为当前正在 CPU 上运行的任务。另一方面是指此任务嵌套调用了新的任务，CPU 正在执行新任务，此任务暂时挂起，等新任务执行完成后 CPU 会回到此任务继续执行，所以此任务马上就会被调度执行了。这种有嵌套调用关系的任务数不只两个，可以很多，比如任务 A 调用了任务 A.1，任务 A.1 又调用了任务 A.1.1 等，为维护这种嵌套调用的关联，CPU 把新任务 TSS 中的 B 位置为 1，并且在新任务的 TSS 中保存了上一级旧任务的 TSS 指针（还要把新任务标志寄存器 eflags 中 NT 位的值置为 1），新老任务的调用关系形成了调用关系链。这里是为了解释繁忙的意义才剧透了任务嵌套，有关这方面内容后面还会有详述。

当任务刚被创建时，此时尚未上 CPU 执行，因此，此时的 B 位为 0，TYPE 的值为 1001。当任务开始上 CPU 执行时，处理器自动地把 B 位置为 1，此时 TYPE 的值为 1011。当任务被换下 CPU 时，处理器把 B 位置 0。注意，B 位是由 CPU 来维护的，不需要咱们人工干预。

B 位存在的意义可不是单纯为了表示任务忙不忙，而是为了给当前任务打个标记，目的是避免当前任务调用自己，也就是说任务是不可重入的。不可重入的意思是当前任务只能调用其他任务，不能自己调用自己。原因是如果任务可以自我调用的话就混乱了，由于旧任务和新任务是同一个，首先 CPU 进行任务状态保护时，在同一个 TSS 中保存后再载入，这将导致严重错误。其次，旧任务在调用新任务时，新任务执行完成后，为了能够回到旧任务，在调用新任务之初，CPU 会自动把老任务的 TSS 选择子写入到新

任务 TSS 中的“上一个任务的 TSS 指针”字段中（后面在任务切换时会讨论），此指针形成了一个任务嵌套调用链，CPU 是靠此指针形成的链表来维护任务调用链的。如果任务重入的话，此链则被破坏。

为避免这种情况的发生，CPU 利用 B 位来判断被调用的任务是否是当前任务，若被调用任务的 B 位为 1，这就表示当前任务自己在调用自己。因此，B 位主要是用来给 CPU 做重入判断用的。

注意，并不是只有当前任务的 B 位才为 1，那些被当前任务通过 call 指令嵌套调用的新任务，除了其 TSS 的 B 位会被置为 1 以外，老任务 TSS 的 B 位不会被清 0，而是继续保持为 1。因为 call 指令是“有去有回”的指令，它执行新任务后还需要再回来，新任务属于当前任务（老任务）的分支。老任务由于未执行完，相当于被自己调用的新任务中断了，因此原任务 TSS 中的 B 位依然保持为 1，并不会被置为 0。

顺便说一句，嵌套任务调用的情况还会影响 eflags 寄存器中的 NT 位，这表示任务嵌套（Nest Tast），后面在介绍任务调度时会细说。

任务是单独的个体，因此每个任务都拥有自己的 TSS。当然，这只是 Intel 这么设想的，现代操作系统为了效率问题，一般并不这么做，后面咱们会说。

TSS 描述符是用来描述 TSS 的，现在介绍下 TSS。

TSS 同其他普通段一样，是位于内存中的区域，因此可以把 TSS 理解为 TSS 段，只不过 TSS 中的数据并不像其他普通段那样散乱，TSS 中的数据是按照固定格式来存储的，所以 TSS 是个数据结构，此结构已经在第 5 章图 5-47 中展示了，为方便学习，再次把该图贴到此处，如图 11-4 所示。

31	15	0
I/O位图在TSS中的偏移地址	（保留） T	100
（保留）	ldt 选择子	96
（保留）	gs	92
（保留）	fs	88
（保留）	ds	84
（保留）	ss	80
（保留）	cs	76
（保留）	es	72
edi		68
esi		64
ebp		60
esp		56
ebx		52
edx		48
ecx		44
eax		40
eflags		36
eip		32
cr3(pdbr)		28
（保留）	SS2	24
esp 2		20
（保留）	SS1	16
esp1		12
（保留）	SS0	8
esp 0		4
（保留）	上一个任务的 TSS 指针	0

▲图 11-4　32 位 TSS 结构

您看，TSS 中的字段基本上全是寄存器名称，这些寄存器就是任务运行中的最新状态。这就像拍照片一样，按下快门的一瞬间，胶片上记录的是事物当时的最新状态，因此也称为快照。可见 TSS 的主要作用就是保存任务的快照，也就是 CPU 执行该任务时，寄存器当时的瞬时值。

除了一般的寄存器外，TSS 中还有“I/O 位图”和“上一个任务的 TSS 指针”，分别位于 TSS 结构图的左上角和右下角。I/O 位图咱们已经在第 5 章中介绍过了，它在单个端口的粒度上进行 IO 特权控制，那这个“上一个任务的 TSS 指针”是干什么的呢？我知道您可能对此很好奇，别着急，现在还不到介绍它的时候，咱们先在此处埋下伏笔，到下一节中为您揭晓答案。

另外要说的就是 TSS 中有三组栈：SS0 和 esp0，SS1 和 esp1，SS2 和 esp2。之前已经介绍过，除了从中断和调用门返回外，CPU 不允许从高特权级转向低特权级（为了助记，可以简单理解为低特权级能做

的高特权级也能做，高特权级不需要找低特权级帮忙）。另外，CPU 在不同特权级下用不同的栈，这三组栈是用来由低特权级往高特权级跳转时用的，最低的特权级是 3，没有更低的特权级会跳入 3 特权级，因此，TSS 中没有 SS3 和 esp3。

需要再次强调的是这三组栈仅仅是 CPU 用来由低特权级进入高特权级时用的，因此，CPU 并不会主动在 TSS 中更新相应特权级的栈指针，不管进入高特权级后进行了多少次压栈操作，下次重新进入该特权级时，该特权级别的栈指针依然是 TSS 中最初的值，除非人为地在 TSS 中将栈指针改写，否则这三组栈指针将一成不变。

Linux 只用到了 0 特权级和 3 特权级，用户进程处于 3 特权级，内核位于 0 特权级，因此对于 Linux 来说只需要在 TSS 中设置 SS0 和 esp0，咱们也效仿它，只设置 SS0 和 esp0 的值就够了。

TSS 是 CPU 原生支持的数据结构，因此 CPU 能够直接、正确识别其中的所有字段。当任务被换下 CPU 时，CPU 会自动将当前寄存器中的值存储到 TSS 中的对应位置，当有新任务上 CPU 运行时，CPU 会自动从新任务的 TSS 中找到相应的寄存器值加载到对应的寄存器中。

也许您要问了，每个任务都有自己的 TSS 结构，而 TSS 是个内存区域，CPU 是怎么知道它在哪里的呢？

和 LDT 一样，CPU 对 TSS 的处理也采取了类似的方式，它提供了一个寄存器来存储 TSS 的起始地址及偏移大小。但也许让人有点意外，这个寄存器不叫 TSSR，而是称为 TR（Task Register），也许是称为 TR，其名称意义也很清晰，而且更为简单吧，当然这是我猜的，至于具体原因是什么不重要，咱们会用就行了。TR 寄存器的结构如图 11-5 所示。

16 位 tss 选择子	32 位线性基址	20 位段界限	属性
选择器	描述符缓冲器		

▲图 11-5　TR 寄存器

TSS 和 LDT 一样，必须要在 GDT 中注册才行，这也是为了在引用描述符的阶段做安全检查。因此 TSS 是通过选择子来访问的，将 tss 加载到寄存器 TR 的指令是 ltr，其指令格式为：

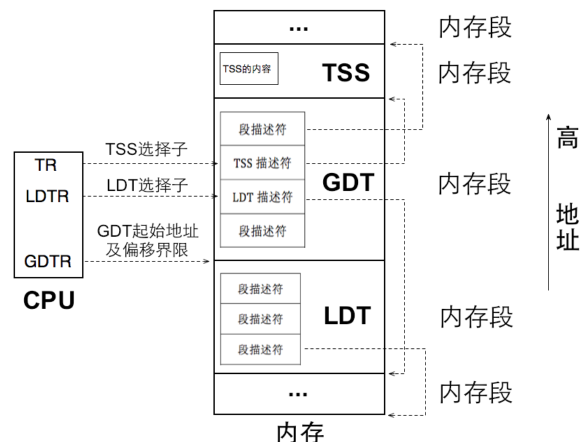
```
ltr  "16 位通用寄存器"或"16 位内存单元"
```

不管操作数是寄存器，还是内存，其值必须是描述符在 GDT 中的选择子。

有了 TSS 后，任务在被换下 CPU 时，由 CPU 自动地把当前任务的资源状态（所有寄存器、必要的内存结构，如栈等）保存到该任务对应的 TSS 中（由寄存器 TR 指定）。CPU 通过新任务的 TSS 选择子加载新任务时，会把该 TSS 中的数据载入到 CPU 的寄存器中，同时用此 TSS 描述符更新寄存器 TR。注意啦，以上动作是 CPU 自动完成的，不需要人工干预，这就是前面所说的硬件一级的原生支持。不过话又说回来了，第一个任务的 TSS 是需要手工加载的，否则第一个任务的状态该没有地方保存了。

总结一下。

TSS 由用户提供，由 CPU 自动维护。



▲图 11-6　TSS、LDT、GDT 全景图

TSS 与其他普通段一样，也有自己的描述符，即 TSS 描述符，用它来描述一个 TSS 的信息，此描述符需要定义在 GDT 中。寄存器 TR 始终指向当前任务的 TSS。任务切换就是改变 TR 的指向，CPU 自动将当前寄存器组的值（快照）写入 TR 指向的 TSS，同时将新任务 TSS 中的各寄存器的值载入 CPU 中对应的寄存器，从而实现了任务切换。

TSS 和 LDT 都只能且必须在 GDT 中注册描述符，TR 寄存器中存储的是 TSS 的选择子，LDTR 寄存器中存储的是 LDT 的选择子，GDTR 寄存器中存储的是 GDT 的起始地址及界限偏移（大小减 1），下面把 TSS 和 LDT 的全景图给大伙儿呈上，如图 11-6 所示。

好，本节到此结束，下节咱们聊聊 CPU 原生支持的任务切换方式。

11.1.4　CPU 原生支持的任务切换方式

本节要介绍的内容是 CPU 厂商原本计划的一种任务切换方法，并不是咱们项目中任务切换的方法，未采用的原因是此方法效率不高，现代操作系统很少用这种方法切换任务。故本节依然属于知识扩展，所以您懂的，可以略过不看。

进入正题，为了支持多任务，CPU 厂商提供了 LDT 及 TSS 这两种原生支持，他们要求为每一个任务分别配一个 LDT 及 TSS（这由咱们程序员来构建），LDT 中保存的是任务自己的实体资源，也就是数据和代码，TSS 中保存的是任务的上下文状态及三种特权级的栈指针、I/O 位图等信息。既然 LDT 和 TSS 用来表示一个任务，那么任务切换就是换这两个结构：将新任务对应的 LDT 信息加载到 LDTR 寄存器，对应的 TSS 信息加载到 TR 寄存器。下面我们看看 CPU 是怎样进行任务切换的。

TSS 被 CPU 用于保存任务的状态及任务状态的恢复，而 LDT 是任务的实体资源，CPU 厂商只是建议这样做，其实没有 LDT 的话也是可以的。比如我们可以把任务自己的段描述符放在 GDT 中，或者干脆采用平坦模型直接用那个 4GB 大小的全局描述符。任务的段放在 GDT，还是 LDT 中，无非就是在用选择子选择它们时有区别，区别您懂的，就是选择子中 TI 位的取值，0 是从 GDT 中选择段描述符，1 是从 LDT 中选择段描述符。描述符及描述符表只是逻辑上对内存区域的划分（当然这也包括其他各种属性，但对此来说并不重要），任务要想执行，归根结底都是用 CS:[E]ip 指向这个任务的代码段内存区域以及 DS 指向其数据段内存区域，所以任务私有的实体资源不是必须放在它自己的 LDT 中。

综上所述，LDT 是可有可无的，真正用于区分一个任务的标志是 TSS，所以用于任务切换的根本方法必然是和任务的 TSS 选择子相关。

进行任务切换的方式有“中断+任务门”，“call 或 jmp+任务门”和 iretd，下面分别介绍。

1．通过“中断+任务门”进行任务切换

其实咱们对采用中断这种方式进行任务切换早已熟悉了，目前的线程切换中用的就是时钟中断。中断是定时发生的，因此用中断进行任务切换的好处是明显的。

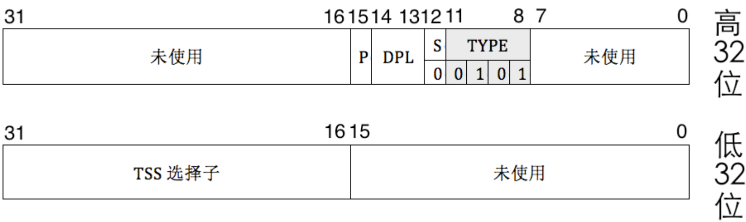
- 实现简单。
- 抢占式多任务调度，所有任务都有运行的机会。

大伙儿回忆一下，在 8259A 中，咱们把时钟的中断向量号设置为 0x20，因此在中断描述符表 IDT 的第 0x20 个中断描述符中注册了时钟的中断处理程序。随着时钟中断的定期发生，满足一定条件后，该中断处理程序又调用 schedule()进行线程调度。

前面说过了，和任务相关的是 TSS 选择子，咱们这个时钟中断的描述符是中断门描述符，中断门中存储的不是 TSS 选择子，而是目标中断处理例程的代码段选择子及偏移地址，因此处理器并没有把此中断门描述符中的中断处理程序当成新的任务。

大伙儿知道，中断发生时，处理器一定会通过中断向量号检索 IDT 中的描述符，所以，若想通过中断的方式进行任务切换，该中断对应的描述符中必须要包含 TSS 选择子，唯一包含 TSS 选择子的描述符便是任务门描述符。

CPU 为原生支持多任务做了很多努力，最直接实现任务切换的方式是任务门。第 5 章的图 5-49 已经展示了任务门描述符，为方便大家学习，这里再将其贴出来，如图 11-7 所示。



▲图 11-7　任务门描述符格式

您看，任务门描述符中的内容是 TSS 选择子，任务门描述符也是系统段，因此 S 的值为 0，在 S 为 0 的情况下，TYPE 的值为 0101 时，就表示此描述符是任务门描述符。

中断是任何时候都会发生的，任务在执行时都会被中断信号打断，在中断描述符表中的描述符可以是中断门、陷阱门、任务门，所以，当前任务被中断后，要么是去执行中断处理程序，要么是进行任务切换。

当中断发生时，处理器通过中断向量号在 IDT 中找到描述符后，通过分析描述符中字段 S 和字段 TYPE 的组合，判断描述符的类型。

若发现此中断对应的描述符是中断门描述符，则转而去执行此中断门描述符中指定的中断处理例程。在中断处理程序的最后，通过 iretd 指令返回到被中断任务的中断前的代码处。

若发现中断对应的是门描述符，此时便进行任务切换，在进一步讨论之前，咱们有必要复习一下。

之前咱们讨论过，一个完整的任务包括用户空间代码及内核空间代码，这两种代码加起来才是任务的全局空间。另外，在 CPU 眼里，一个 TSS 就代表一个任务，TSS 才是任务的标志，CPU 区分任务就是靠 TSS，因此，只要 TR 寄存器中的 TSS 信息不换，无论执行的是哪里的指令，也无论指令是否跨越特权级（从用户态到内核态），CPU 都认为还是在同一个任务中。

咱们平时所写的程序代码都只是用户态代码，对于完整的任务来说它属于半成品。用户代码和内核代码只是同一个任务的不同部分而已。中断处理例程属于内核代码，因此它也属于当前的任务，当在中断处理例程中执行 iretd 指令从中断返回后，是返回到当前任务在中断前的代码处，依然属于当前任务，只是返回到了当前任务的不同部分。

之前咱们接触中断时，我们已了解 iretd 指令用于从中断处理例程中返回，其实这只是它的一个功能，它一共有两个功能。

（1）从中断返回到当前任务的中断前代码处。

（2）当前任务是被嵌套调用时，它会调用自己 TSS 中“上一个任务的 TSS 指针”的任务，也就是返回到上一个任务。

第 2 个功能是咱们现在关注的重点：iretd 可以调用一个任务。

一个指令在不同环境下具备不同的功能，有时候这很容易引起混淆，现在模拟一下这个情况就知道了，当中断发生时，假设当前任务 A 被中断，CPU 进入中断后，它有可能的动作是：

- 假设是中断门或陷阱门，执行完中断处理例程后是用 iretd 指令返回到任务 A 中断前的指令部分。
- 假设是任务门，进行任务切换，此时是嵌套调用任务 B，任务 B 在执行期间又发生了中断，进入了对应的中断门，当执行完对应的中断处理程序后，用 iretd 指令返回。
- 同样假设是任务门，任务 A 调用任务 B 执行，任务 B 执行完成后要通过 iretd 指令返回到任务 A，使任务 A 继续完成后续的指令。

以上几种情况的最后都是执行 iretd 指令，那么，CPU 在执行 iretd 时，是回到任务 A 中断前的代码部分，还是回到任务 B 中断前的代码部分？还是调用任务 A 呢？必须要分清楚这几种情况，因为这涉及的底层操作不同。

怎么区分这几种情况呢？

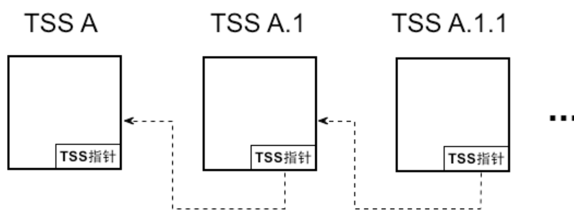
看来必须在调用新任务之初就给自己留好“后路”，这时候标志寄存器 eflags 中的 NT 位和 TSS 中的“上一个任务的 TSS 指针”字段便起作用了。

NT 位是 eflags 中的第 14 位，1bit 的宽度，它表示 Nest Task Flag，任务嵌套。任务嵌套是指当前任务是被前一个任务调用后才执行的，也就是当前任务嵌套于另一个任务中，相当于另一个任务的子任务，在此任务执行完成后还要回到前一个任务，使其继续执行。这一点类似于我们在函数 A 中调用一个子函数 Ac，子函数 Ac 执行完成后还是要回到函数 A 中。

TSS 的字段“上一个任务的 TSS 指针”，用于记录是哪个任务调用了当前任务，有些类似于“父任务”，此字段中的值是 TSS 的地址，因此它就形成了任务嵌套关系的单向链表，每个 TSS 属于链表中的结点，CPU 用此链表来记录任务的嵌套调用关系，如图 11-8 所示。

当调用一个新任务时，处理器做了两件准备工作。

- 自动将新任务 eflags 中的 NT 位置为 1，这就表示新任务能够执行的原因是被别的任务调用，也就是嵌套调用。
- 随后处理器将旧任务的 TSS 选择子写入新任务 TSS 的“上一个任务的 TSS 指针”字段中。



▲图 11-8　TSS 链表

一起回忆一下，中断发生时，处理器要把 NT 位和 TF 位置为 0，如果对应的描述符是中断门描述符，还要再将标志寄存器 eflags 中的 IF 位清 0，这是为了避免中断嵌套，防止正在处理的中断尚未完成时相同的中断源又发出中断信号，避免引发 GP 异常。

不管处理器把标志寄存器 eflags 中的值修改成什么样，这都是任务进入中断后的事，任务 eflags 中原本的值早已经在进入中断前压入栈保存了，iretd 退出中断时这些值会被恢复。

有了上面的准备工作后，当 CPU 执行 iretd 指令时，始终要判断 NT 位的值。如果 NT 等于 1，这表示是从新任务返回到旧任务，于是 CPU 到当前任务（新任务）TSS 的“上一个任务的 TSS 指针”字段中获取旧任务的 TSS，转而去执行旧任务。如果 NT 等于 0，这表示要回到当前任务中断前的指令部分。

强调一下，只有 NT 为 1 时，iretd 的功能才是任务调用（返回到旧任务）。

综上所述，中断发生时，通过任务门进行任务切换的过程如下。

（1）从该任务门描述符中取出任务的 TSS 选择子。

（2）用新任务的 TSS 选择子在 GDT 中索引 TSS 描述符。

（3）判断该 TSS 描述符的 P 位是否为 1，为 1 表示该 TSS 描述符对应的 TSS 已经位于内存中 TSS 描述符指定的位置，可以访问。否则 P 不为 1 表示该 TSS 描述符对应的 TSS 不在内存中，这会导致异常。

（4）从寄存器 TR 中获取旧任务的 TSS 位置，保存旧任务（当前任务）的状态到旧 TSS 中。其中，任务状态是指 CPU 中寄存器的值，这仅包括 TSS 结构中列出的寄存器：8 个通用寄存器，6 个段寄存器，指令指针 eip，栈指针寄存器 esp，页表寄存器 cr3 和标志寄存器 eflags 等。

（5）把新任务的 TSS 中的值加载到相应的寄存器中。

（6）使寄存器 TR 指向新任务的 TSS。

（7）将新任务（当前任务）的 TSS 描述符中的 B 位置 1。

（8）将新任务标志寄存器中 eflags 的 NT 位置 1。

（9）将旧任务的 TSS 选择子写入新任务 TSS 中“上一个任务的 TSS 指针”字段中。

（10）开始执行新任务。

在执行新任务之前，旧任务是当前的任务，因此旧任务 TSS 描述符中的 B 位为 1，在调用新任务后也不会修改，因为它尚未执行完，属于嵌套调用别的任务，并不是单独的任务。

当新任务执行完成后，调用 iretd 指令返回到旧任务，此时处理器检查 NT 位，若其值为 1，便进行返回工作，步骤如下。

（1）将当前任务（新任务）标志寄存器中 eflags 的 NT 位置 0。

（2）将当前任务 TSS 描述符中的 B 位置为 0。

（3）将当前任务的状态信息写入 TR 指向的 TSS。

（4）获取当前任务 TSS 中“上一个任务的 TSS 指针”字段的值，将其加载到 TR 中，恢复上一个任务的状态。

（5）执行上一个任务（当前任务），从而恢复到旧任务。

2．call、jmp 切换任务

开门见山：

（1）首先，任务门描述符除了可以在 IDT 中注册，还可以在 GDT 和 LDT 中注册。

（2）其次，任务以 TSS 为代表，只要包括 TSS 选择子的对象都可以作为任务切换的操作数。

因此另一种切换任务的方式是用 call 和 jmp 指令+TSS 选择子或任务门选择子。

TSS 中已经包含了任务的详细信息，任务门描述符中又包含了 TSS 选择子（似乎看来任务门很多余），所以它和所有的门描述符一样，使用 TSS 和任务门作为 call 和 jmp 指令操作数时，操作数中包含了偏移量，CPU 只用选择子部分就够了，会忽略其中的偏移量部分。

假设任务门选择子定义在 GDT 中第 2 个描述符位置：

```
call 0x0010:0x1234
```

假设 TSS 选择子定义在 GDT 中第 3 个描述符位置：

```
call 0x0018:0x1234
```

上述两个指令中的偏移量 0x1234 都会被处理器忽略。

call 是有去有回的指令，jmp 是一去不回的指令，它们在调用新任务时的区别也在于此。

call 指令以任务嵌套的方式调用新任务，当以 call 指令调用新任务时，我们以操作数为 TSS 选择子为例，比如“call 0x0018:0x1234”，任务切换的步骤如下。

（1）CPU 忽略偏移量 0x1234，拿选择子 0x0018 在 GDT 中索引到第 3 个描述符。

（2）检查描述符中的 P 位，若 P 为 0，表示该描述符对应的段不存在，这将引发异常。同时检查该描述符的 S 与 TYPE 的值，判断其类型，如果是 TSS 描述符，检查该描述符的 B 位，B 位若为 1 将抛出 GP 异常，即表示调用不可重入。

（3）进行特权级检查，数值上“CPL 和 TSS 选择子中的 RPL”都要小于等于 TSS 描述符的 DPL，否则抛出 GP 异常。

（4）特权检查完成后，将当前任务的状态保存到寄存器 TR 指向的 TSS 中。

（5）加载新任务 TSS 选择子到 TR 寄存器的选择器部分，同时把 TSS 描述符中的起始地址和偏移量等属性加载到 TR 寄存器中的描述符缓冲器中。

（6）将新任务 TSS 中的寄存器数据载入到相应的寄存器中，同时进行特权级检查，如果检查未通过，则抛出 GP 异常。

（7）CPU 会把新任务的标志寄存器 eflags 中的 NT 位置为 1。

（8）将旧任务 TSS 选择子写入新任务 TSS 中的字段“上一个任务的 TSS 指针”中，这表示新任务是被旧任务调用才执行的。

（9）然后将新任务 TSS 描述符中的 B 位置为 1 以表示任务忙。旧任务 TSS 描述符中的 B 位不变，依然保持为 1，旧任务的标志寄存器 eflags 中的 NT 位的值保持不变，之前是多少就是多少。

（10）开始执行新任务，完成任务切换。

jmp 指令以非嵌套的方式调用新任务，新任务和旧任务之间不会形成链式关系。当以 jmp 指令调用新任务时，新任务 TSS 描述符中的 B 位会被 CPU 置为 1 以表示任务忙，旧任务 TSS 描述符中的 B 位会被 CPU 清 0。

当通过 iretd 指令任务返回时，新任务 eflags 寄存器的 NT 位必须为 1，所以 iretd 仅适用于 call。当调用 iretd 返回到旧任务时，CPU 会将当前任务（新任务）TSS 描述符中 B 位清 0，同时将其 eflags 寄存器的 NT 位清 0。

好啦，以上介绍的是 CPU 原生支持的任务切换方式，因为效率问题，现代操作系统如 Linux 未采用此方式，咱们也未采用它们，所以相关内容就介绍到这里，大伙儿有兴趣的话还是自行查阅介绍 Intel 的三卷手册吧。

11.1.5 现代操作系统采用的任务切换方式

TSS 是 x86 CPU 的特定结构，被用来定义“任务”，它是内置到处理器原生支持的多任务的一种形式。

上节中，我们介绍了 CPU 提供的多任务支持，每个任务拥有自己的 TSS，每个任务也可以有自己的 LDT，看样子还是很简洁的，但为什么 Linux 未采用此方式呢？

首先，在上一节中，我们用“call 0x0018:0x1234”举例说明了通过 call 指令+TSS 选择子的形式进行任务切换的过程。您看，此过程大概分成 10 步，这还是直接用 TSS 选择子进行任务切换的步骤，这已经非常繁琐了，在每一次任务切换过程中，CPU 除了做特权级检查外，还要在 TSS 的加载、保存、设置 B 位，以及设置标志寄存器 eflags 的 NT 位诸多方面消耗很多精力，这导致此种切换方式效能很低。

其次，常见的指令集有两大派系，复杂指令集 CISC 和精简指令集 RISC。x86 使用的指令集属于 CISC，在此指令集的发展过程中，工程师为了让程序员少写代码，把指令的功能做得越发强大，因此在 RISC 中多条指令才能完成的工作，在 CISC 中只用一条指令就完成了。看上去感觉很爽的样子，但这只是开发效率上的提升，执行效率却下降了，原因是：表面强大的功能是用内部复杂、数量更多的微操作换来的，也就是说，CISC 的强大需要更多的时钟周期作为代价。虽然 Intel 提供了 call 和 jmp 指令实现任务切换，但这两个指令所消耗的时钟周期也是可观的，都是以百为单位的（据说已经达到 300+，我没测试过）。

最后，一个任务需要单独关联一个 TSS，TSS 需要在 GDT 中注册，GDT 中最多支持 8192 个描述符，为了支持更多的任务，随着任务的增减，要及时修改 GDT，在其中增减 TSS 描述符，修改过后还要重新加载 GDT。这种频繁修改描述符表的操作还是很消耗 CPU 资源的。

以上是效率方面的原因，除了效率以外，还有便携性和灵活性等原因，不仅 Linux 未采用这种原生的任务切换方法，而且几乎所有 x86 操作系统都未采用。

不幸的是，我们是在 CPU 制定的规则上编写程序的，因此始终脱离不了大规则的束缚，还有一件工作必须且只能用 TSS 来完成，这就是 CPU 向更高特权级转移时所使用的栈地址，需要提前在 TSS 中写入。

导致转移到更高特权级的一种情况是在用户模式下发生中断，CPU 会由低特权级进入高特权级，这会发生堆栈的切换。当一个中断发生在用户模式（特权级 3），处理器从当前 TSS 的 SS0 和 esp0 成员中获取用于处理中断的堆栈。因此，我们必须创建一个 TSS，并且至少初始化 TSS 中的这些字段。

尽管 CPU 提供了 0、1、2、3 共 4 个特权级，但我们效仿 Linux 只用其中的 2 个，内核处理特权级 0，用户进程处于特权级 3。

结论：我们使用 TSS 唯一的理由是为 0 特权级的任务提供栈。

咱们是效仿 Linux 的任务切换方法的，那有必要看看 Linux 是怎样做的。

硬件是软件的舞台，软件再强大也要向硬件 CPU 低头，CPU 要求用 TSS 这是硬指标，Linux 也得遵守。不过为了“应付”这一指标，Linux 为每个 CPU 创建一个 TSS，在各个 CPU 上的所有任务共享同一个 TSS，各 CPU 的 TR 寄存器保存各 CPU 上的 TSS，在用 ltr 指令加载 TSS 后，该 TR 寄存器永远指向同一个 TSS，之后再也不会重新加载 TSS。在进程切换时，只需要把 TSS 中的 SS0 及 esp0 更新为新任务的内核栈的段地址及栈指针。

您看，实际上 Linux 对 TSS 的操作是一次性加载 TSS 到 TR，之后不断修改同一个 TSS 的内容，不再进行重复加载操作。

Linux 在 TSS 中只初始化了 SS0、esp0 和 I/O 位图字段，除此之外 TSS 便没用了，就是个空架子，不再做保存任务状态之用。

那任务的状态信息保存在哪里呢？

是这样的，当 CPU 由低特权级进入高特权级时，CPU 会“自动”从 TSS 中获取对应高特权级的栈指针（TSS 是 CPU 内部框架原生支持的嘛，当然是自动从中获取新的栈指针）。我们具体说一下，Linux 只用到了特权 3 级和特权 0 级，因此 CPU 从 3 特权级的用户态进入 0 特权级的内核态时（比如从用户进程进入中断），CPU 自动从当前任务的 TSS 中获取 SS0 和 esp0 字段的值作为 0 特权级的栈，然后 Linux“手动”执行一系列的 push 指令将任务的状态的保存在 0 特权级栈中，也就是 TSS 中 SS0 和 esp0 所指向的栈。

要知道，人家 Intel 当初是打算让 TR 寄存器指向不同任务的 TSS 以实现任务切换的，Linux 这里只换了 TSS 中的部分内容，而 TR 本身没换，还是指向同一个 TSS，这种“自欺欺人”的好处是任务切换的开销更小了，因为和修改 TSS 中的内容所带来的开销相比，在 TR 中加载 TSS 的开销要大得多。您想，每次切换任务都要用 ltr 指令重新加载新任务的 TSS 到寄存器 TR，TSS 是位于内存中的，而内存很慢的，随着任务数量一多，这种频繁重复加载的开销就更为“可观”。

另外，Linux 中任务切换不使用 call 和 jmp 指令，这也避免了任务切换的低效（想想在上一节中通过“call+TSS 选择子”任务切换的 10 个步骤，又是来回做特权检查，又是更新 TSS 中的 B 位及上一个任务 TSS 的指针，还要设置 eflags 标志位，是不是很繁琐）。

综上所述，Linux 的任务切换效率比 CPU 原生方案大大提升，咱们也参照 Linux 的做法实现咱们的任

务切换。

本节到此结束，在下一节中咱们将定义 TSS 并将其初始化，为实现用户做好基础工作，好啦，兄弟们下节咱们再叙。

11.2 定义并初始化 TSS

怎么感觉好久没写代码了呢，哈哈，终于到了这一节，本节咱们准备迈向用户进程的第一步，创建 TSS。

该说的前面都已经说不少了，直接上菜，代码 11-1 是我们本节在 global.h 中增加的属性，它们会在代码 11-2 中用到，大伙儿先粗略过一下。

代码 11-1　（project/c11/a/kernel/global.h）

```
 6 // ----------------  GDT 描述符属性  ----------------
 7
 8 #define DESC_G_4K       1
 9 #define DESC_D_32       1
10 #define DESC_L          0  // 64 位代码标记，此处标记为 0 便可
11 #define DESC_AVL        0  // cpu 不用此位，暂置为 0
12 #define DESC_P          1
13 #define DESC_DPL_0      0
14 #define DESC_DPL_1      1
15 #define DESC_DPL_2      2
16 #define DESC_DPL_3      3
17 /****************************************************************
18    代码段和数据段属于存储段，tss 和各种门描述符属于系统段
19    s 为 1 时表示存储段，为 0 时表示系统段
20 ****************************************************************/
21 #define DESC_S_CODE     1
22 #define DESC_S_DATA     DESC_S_CODE
23 #define DESC_S_SYS      0
24 #define DESC_TYPE_CODE  8
// x=1,c=0,r=0,a=0 代码段是可执行的、非依从的、不可读的，已访问位 a 清 0

25 #define DESC_TYPE_DATA  2
// x=0,e=0,w=1,a=0 数据段是不可执行的、向上扩展的、可写的，已访问位 a 清 0

26 #define DESC_TYPE_TSS   9      // B 位为 0，不忙

…略
37 #define SELECTOR_K_CODE    ((1 << 3) + (TI_GDT << 2) + RPL0)
38 #define SELECTOR_K_DATA    ((2 << 3) + (TI_GDT << 2) + RPL0)
39 #define SELECTOR_K_STACK   SELECTOR_K_DATA
40 #define SELECTOR_K_GS     ((3 << 3) + (TI_GDT << 2) + RPL0)
41 /* 第 3 个段描述符是显存，第 4 个是 tss */
42 #define SELECTOR_U_CODE    ((5 << 3) + (TI_GDT << 2) + RPL3)
43 #define SELECTOR_U_DATA    ((6 << 3) + (TI_GDT << 2) + RPL3)
44 #define SELECTOR_U_STACK   SELECTOR_U_DATA
45
46 #define GDT_ATTR_HIGH    \
((DESC_G_4K << 7) + (DESC_D_32 << 6) + (DESC_L << 5) + (DESC_AVL << 4))

47 #define GDT_CODE_ATTR_LOW_DPL3 \
((DESC_P << 7) + \
(DESC_DPL_3 << 5) + \
(DESC_S_CODE << 4) +
DESC_TYPE_CODE)
48 #define GDT_DATA_ATTR_LOW_DPL3 \
((DESC_P << 7) + \
(DESC_DPL_3 << 5) + \
(DESC_S_DATA << 4) +
DESC_TYPE_DATA)
49
50
51 //---------------  TSS 描述符属性  ------------
52 #define TSS_DESC_D  0
53
54 #define TSS_ATTR_HIGH \
```

```
((DESC_G_4K << 7) + \
(TSS_DESC_D << 6) + \
(DESC_L << 5) + \
(DESC_AVL << 4) + 0x0)

55 #define TSS_ATTR_LOW \
   ((DESC_P << 7) + \
   (DESC_DPL_0 << 5) +\
    (DESC_S_SYS << 4) + \
   DESC_TYPE_TSS)

56 #define SELECTOR_TSS ((4 << 3) + (TI_GDT << 2 ) + RPL0)
57
58 /* 定义 GDT 中描述符的结构 */
59 struct gdt_desc {
60    uint16_t limit_low_word;
61    uint16_t base_low_word;
62    uint8_t  base_mid_byte;
63    uint8_t  attr_low_byte;
64    uint8_t  limit_high_attr_high;
65    uint8_t  base_high_byte;
66 };
```

好啦，匆匆略过后，咱们要看看今天的主题——tss.c，我们新建一个目录 userprog，今后有关用户进程的代码文件都将存放在此目录中。下面请看代码 11-2。

代码 11-2　（project/c11/a/userprog/tss.c）

```
…略
 7 /* 任务状态段 tss 结构 */
 8 struct tss {
 9      uint32_t backlink;
10      uint32_t* esp0;
11      uint32_t ss0;
12      uint32_t* esp1;
13      uint32_t ss1;
14      uint32_t* esp2;
15      uint32_t ss2;
16      uint32_t cr3;
17      uint32_t (*eip) (void);
18      uint32_t eflags;
19      uint32_t eax;
20      uint32_t ecx;
21      uint32_t edx;
22      uint32_t ebx;
23      uint32_t esp;
24      uint32_t ebp;
25      uint32_t esi;
26      uint32_t edi;
27      uint32_t es;
28      uint32_t cs;
29      uint32_t ss;
30      uint32_t ds;
31      uint32_t fs;
32      uint32_t gs;
33      uint32_t ldt;
34      uint32_t trace;
35      uint32_t io_base;
36 };
37 static struct tss tss;
38
39 /* 更新 tss 中 esp0 字段的值为 pthread 的 0 级线 */
40 void update_tss_esp(struct task_struct* pthread) {
41    tss.esp0 = (uint32_t*)((uint32_t)pthread + PG_SIZE);
42 }
43
44 /* 创建 gdt 描述符 */
45 static struct gdt_desc make_gdt_desc(\
uint32_t* desc_addr, \
uint32_t limit, \
uint8_t attr_low, \
uint8_t attr_high) {
```

```
46     uint32_t desc_base = (uint32_t)desc_addr;
47     struct gdt_desc desc;
48     desc.limit_low_word = limit & 0x0000ffff;
49     desc.base_low_word = desc_base & 0x0000ffff;
50     desc.base_mid_byte = ((desc_base & 0x00ff0000) >> 16);
51     desc.attr_low_byte = (uint8_t)(attr_low);
52     desc.limit_high_attr_high = \
       (((limit & 0x000f0000) >> 16) + (uint8_t)(attr_high));
53     desc.base_high_byte = desc_base >> 24;
54     return desc;
55 }
56
57 /* 在 gdt 中创建 tss 并重新加载 gdt */
58 void tss_init() {
59     put_str("tss_init start\n");
60     uint32_t tss_size = sizeof(tss);
61     memset(&tss, 0, tss_size);
62     tss.ss0 = SELECTOR_K_STACK;
63     tss.io_base = tss_size;
64
65 /* gdt 段基址为 0x900，把 tss 放到第 4 个位置，也就是 0x900+0x20 的位置 */
66
67    /* 在 gdt 中添加 dpl 为 0 的 TSS 描述符 */
68    *((struct gdt_desc*)0xc0000920) = make_gdt_desc(\
     (uint32_t*)&tss, \
      tss_size - 1, \
      TSS_ATTR_LOW,
                                                 TSS_ATTR_HIGH\
                                                 );
69
70   /* 在 gdt 中添加 dpl 为 3 的数据段和代码段描述符 */
71  *((struct gdt_desc*)0xc0000928) = make_gdt_desc(\
    (uint32_t*)0,\
    0xfffff, \
    GDT_CODE_ATTR_LOW_DPL3, \
                                                 GDT_ATTR_HIGH\
                                                 );

72  *((struct gdt_desc*)0xc0000930) = make_gdt_desc(\
    (uint32_t*)0,\
    0xfffff, \
    GDT_DATA_ATTR_LOW_DPL3, \
                                                 GDT_ATTR_HIGH\
                                                 );

73
74   /* gdt 16 位的 limit 32 位的段基址 */
75    uint64_t gdt_operand = \
      ((8 * 7 - 1) | ((uint64_t)(uint32_t)0xc0000900 << 16));   // 7 个描述符大小

76    asm volatile ("lgdt %0" : : "m" (gdt_operand));
77    asm volatile ("ltr %w0" : : "r" (SELECTOR_TSS));
78    put_str("tss_init and ltr done\n");
79 }
```

在 tss.c 的开头第 8～36 行定义了 TSS 的结构体 struct tss，这是按照前面所介绍的 TSS 结构来定义的，还记得之前说过的吗？TSS 是程序员提供，由 CPU 来维护的，咱们定义好后，在第 37 行实例化一个 tss，一会儿就将此实例交给 CPU。

第 40 行定义的函数 update_tss_esp 用来更新 TSS 中的 esp0，这是学习 Linux 任务切换的方式，只修改 TSS 中的特权级 0 对应的栈。此函数将 TSS 中 esp0 修改为参数 pthread 的 0 级栈地址，也就是线程 pthread 的 PCB 所在页的最顶端——(uint32_t)pthread + PG_SIZE。此栈地址是用户进程由用户态进入内核态时所用的栈，这和之前咱们的内核线程地址是一样的，也许您猜到了，用户进程进入内核态后，除了拥有单独的地址空间外，其他方面和内核线程是一样的。这一点在进一步实现用户进程时会更有体会。

我们的 GDT 是在 loader.S 中进入保护模式时实现的，那时候我们还是以小米加步枪的方式，用汇编语言直接生成的 GDT。进入内核后，我们已经用 C 语言编程了，现在我们还需要 GDT 中增加新的描述符，

因此在第 45 行创建了函数 make_gdt_desc，专门生成描述符结构，注意此函数并不是直接在 GDT 中安装好描述符，只是返回生成的描述符。此函数的实现是按照段描述符的格式来拼数据，在内部生成一局部描述符结构体变量 struct gdt_desc desc，后面把此结构体变量中的属性填充好后通过 return 返回其值。

第 58 行是函数 tss_init，此函数除了用来初始化 tss 并将其安装到 GDT 中外，还另外在 GDT 中安装两个供用户进程使用的描述符，一个是 DPL 为 3 的数据段，另一个是 DPL 为 3 的代码段。

第 61 行将全局变量 tss 清 0 后，在第 62 行为其 ss0 字段赋 0 级栈段的选择子 SELECTOR_ K_STACK。

第 63 行代码"tss.io_base = tss_size"，将 tss 的 io_base 字段置为 tss 的大小 tss_size，这表示此 TSS 中并没有 IO 位图。有关 IO 位图的内容咱们已经在第 5 章的 IO 特权级中介绍过了，当 IO 位图的偏移地址大于等于 TSS 大小减 1 时，就表示没有 IO 位图。

在第 68 行，我们在 GDT 中安装 TSS 描述符。在调用 make_gdt_desc 后，其返回的描述符是安装在 0xc0000920 的地址，即*((struct gdt_desc*)0xc0000920)，其实此处用 0x920 也是可以的，还记得吗？我们把低端 1MB 空间的页表映射为同物理地址相同，并且把内核开始使用的第 768 个页表指向了同低端 1MB 空间相同的物理页，因此此时的 0xc0000920 可以用 0x920 代替。

为什么把 TSS 描述符放在 0xc0000920 的地址呢？

复习一下，32 位保护模式下的描述符大小都是 8 字节，在 GDT 中第 0 个段描述符不可用，第 1 个为代码段，第 2 个为数据段和栈，第 3 个为显存段，因此把 tss 放到第 4 个位置，也就是 0xc0000900+0x20 的位置。

接下来在第 71 行和第 72 行安装了两个 DPL 为 3 的段描述符，分别是代码段和数据段，这是为用户进程提前做的准备，它们在 GDT 中的位置基于 TSS 描述符顺延，分别是偏移 GDT0x28 和 0x30 的位置。

万事俱备之后，由于已经变更了 GDT，故需要用 lgdt 指令重新加载 GDT。

在第 75 行，定义了变量 gdt_operand 作为 lgdt 指令的操作数。回想一下 lgdt 的指令格式，其操作数是"16 位表界限&32 位表的起始地址"，这里要求表界限要放在前面，也就是操作数中前 2 字节的低地址处。到目前为止，在原有描述符的基础上我们又新增了 3 个描述符，加上第 0 个不可用的哑描述符，GDT 中现在一共是 7 个描述符，因此表界限值为 8 * 7 - 1。操作数中的高 32 位是 GDT 起始地址，在这里我们把 GDT 线性地址 0xc0000900 先转换成 uint32_t 后，再将其转换成 uint64_t 位（不可一步到位转为 uint64_t），最后通过按位或运算符'|'拼合在一起。

通过内联汇编，第 76 行将新的 GDT 重新加载，第 77 行将 tss 加载到 TR。至此，新的 GDT 和 TSS 已经生效，本节目标已经完成。

将 tss_init 加入 init_all 后，编译运行，结果如图 11-9 所示。

```
I am kernel
init_all
idt_init start
   idt_desc_init done
   pic_init done
idt_init done
mem_init start
   mem_pool_init start
      kernel_pool_bitmap_start:C009A000 kernel_pool_phy_addr_start:200000
      user_pool_bitmap_start:C009A1E0 user_pool_phy_addr_start:1100000
   mem_pool_init done
mem_init done
thread_init start
thread_init done
timer_init start
timer_init done
keyboard init start
keyboard init done
tss_init start
tss_init and ltr done
CTRL + 3rd button enables mouse  NUM  CAPS  SCRL  HD:0-M
```

▲图 11-9　tss 加载成功

之前说过，tss 是交给 CPU 来维护的，在 tss 加载成功后，我们看下 CPU 对 tss 都"做了些什么"，如图 11-10 所示。

您看，在 GDT 中第 4 个描述符是刚安装的 TSS，其显示为 32-Bit TSS (Busy)，这说明 TSS 的 B 被 CPU 置为 1 了，TSS 已经生效。

```
work@localhost:~/my_workspace/bochs
File Edit View Search Terminal Help
00000000000i[     ] using log file bochs.out
Next at t=0
(0) [0x0000fffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ; ea5be000
f0
<bochs:1> c
^CNext at t=108222702
(0) [0x00000000151f] 0008:c000151f (unk. ctxt): jmp .-2 (0xc000151f)      ; ebfe
<bochs:2> info gdt
Global Descriptor Table (base=0xc0000900, limit=55):
GDT[0x00]=??? descriptor hi=0x00000000, lo=0x00000000
GDT[0x01]=Code segment, base=0x00000000, limit=0xffffffff, Execute-Only, Non-Con
forming, Accessed, 32-bit
GDT[0x02]=Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed
GDT[0x03]=Data segment, base=0xc00b8000, limit=0x00007fff, Read/Write, Accessed
GDT[0x04]=32-Bit TSS (Busy) at 0xc0005320, length 0x0006b
GDT[0x05]=Code segment, base=0x00000000, limit=0xffffffff, Execute-Only, Non-Con
forming, 32-bit
GDT[0x06]=Data segment, base=0x00000000, limit=0xffffffff, Read/Write
You can list individual entries with 'info gdt [NUM]' or groups with 'info gdt [
NUM] [NUM]'
<bochs:3> c
^CNext at t=143881290
(0) [0x00000000151f] 0008:c000151f (unk. ctxt): jmp .-2 (0xc000151f)      ; ebfe
<bochs:4>
```

▲图 11-10 TSS busy

好啦，本节为下一步实现用户进程打下了基础，下一节中我们尝试实现用户进程。

11.3 实现用户进程

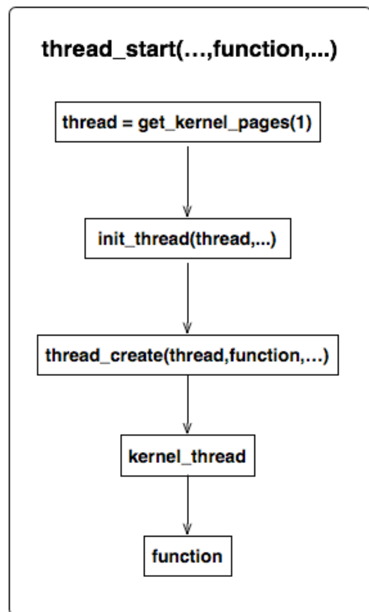
在很久以前我们已经实现了内核线程，本节我们将在线程的基础上实现进程，因此额外的工作量并不大。通过本节的介绍，您将会对进程的本质有所了解。

为了实现进程，我们还要提前准备一些功能，下面先在局部上入手，最后再给大家从整体上梳理。

局部讲解虽然使大家不易掌握整个来龙去脉，但这是了解细节必然的过程，除非您只想了解个框架，但这并不是本书的初衷。因此，在下面的局部功能介绍时，如果您感到迷茫，先不要急，等到我介绍完整个用户进程的部分，您会在整体上贯穿整个过程，到时候自然能在全局上一目了然。

11.3.1 实现用户进程的原理

我们的目标是在现有线程的基础上实现进程，先回忆下，我们创建线程是通过 thread_start 进行的，其内部实现的流程如图 11-11 所示。



▲图 11-11 线程创建流程

在 thread_start（...,function,...）的调用中，function 是我们最终在线程中执行的函数。在 thread_start 内部，先是通过 get_kernel_pages(1) 在内核内存池中获取 1 个物理页做线程的 pcb，即 thread，接着调用 init_thread 初始化该线程 pcb 中的信息，然后再用 thread_create 创建线程运行的栈，实际上是将栈中的返回地址指向了 kernel_thread 函数，因此相当于调用了 kernel_thread，在 kernel_thread 中通过调用 function 的方式使 function 得到执行。

经过以上的分析，如果要基于线程实现进程，我们把 function 替换为创建进程的新函数就可以啦，先把控制权拿到手再说，进程相关的具体工作再由新函数完成。

11.3.2 用户进程的虚拟地址空间

进程与内核线程最大的区别是进程有单独的 4GB 空间，这指的是虚拟地址，物理地址空间可未必有那么大，看似无限的虚拟地址经过分页机制之后，最终要落到有限的物理页中。

每个进程都拥有 4GB 的虚拟地址空间，虚拟地址连续而物理地址可以不连续，这就是保护模式下分

页机制的优势。为演示此特性，我们需要单独为每个进程维护一个虚拟地址池，用此地址池来记录该进程的虚拟中，哪些已被分配，哪些可以分配。

与各个进程相关的数据，如果数据量不大的话，最好是存储在该进程的 PCB 中，这样便于管理。在上一节中您已经知道，进程是基于线程实现的，因此它和线程一样使用相同的 pcb 结构，即 struct task_struct，我们要做的就是在此结构中增加一个成员，用它来跟踪用户空间虚拟地址的分配情况。

具体新增的成员名是在 thread.h 中增加的，如代码 11-3 所示。

代码 11-3 （project/c11/b/thread/thread.h）

```
  3 #include "stdint.h"
  4 #include "list.h"
  5 #include "bitmap.h"
  6 #include "memory.h"
…略
75 /* 进程或线程的 pcb，程序控制块 */
76 struct task_struct {
77     uint32_t* self_kstack;        // 各内核线程都用自己的内核栈
78     enum task_status status;
79     char name[16];
80     uint8_t priority;
…略
87 /* general_tag 的作用是用于线程在一般的队列中的结点 */
88     struct list_elem general_tag;
89
90 /* all_list_tag 的作用是用于线程队列 thread_all_list 中的结点 */
91     struct list_elem all_list_tag;
92
93     uint32_t* pgdir;              // 进程自己页表的虚拟地址
94     struct virtual_addr userprog_vaddr;   // 用户进程的虚拟地址
95     uint32_t stack_magic;
// 用这串数字做栈的边界标记，用于检测栈的溢出
96 };
```

其中第 95 行的 struct virtual_addr userprog_vaddr 便是每个用户进程的虚拟地址池。

顺便说一下，第 93 行的 pgdir 用于存放进程页目录表的虚拟地址，这将在为进程创建页表时为其赋值。

也许您感到迷惑，按理说页表寄存器 cr3 中的应该是页目录表的物理地址，但成员 pgdir 是虚拟地址，这是什么原因？

原因是页目录表本身也要占用内存来存储，我们在为进程创建页目录表时，必然要为其申请内存，但内存管理系统返回的地址肯定都是虚拟地址，不可能返回物理地址，因为返回物理地址也没用，在分页机制下，引用的任何地址都被当作虚拟地址，该“物理地址”也要再次被转换成别的物理地址，这就错了。因此在往寄存器 cr3 中加载页目录地址时，我们会将 pgdir 转换成物理地址，这部分将在下节介绍。

11.3.3 为进程创建页表和 3 特权级栈

进程与线程的区别是进程拥有独立的地址空间，不同的地址空间就是不同的页表，因此我们在创建进程的过程中需要为每个进程单独创建一个页表。我们这里所说的页表是“页目录表+页表”，页目录表用来存放页目录项 PDE，每个 PDE 又指向不同的页表。

页表虽然用于管理内存，但它本身也要用内存来存储，所以要为每个进程单独申请存储页目录项及页表项的虚拟内存页。

除此之外，咱们之前创建的线程属于内核的线程，它们运行在特权级 0。和它们相比，用户进程还多了个特权级 3，大多数情况下，用户进程在特权级 3 下工作，因此，我们还要为用户进程创建在 3 特权级的栈。栈也是内存区域，所以，咱们还得为进程分配内存（虚拟内存）作为 3 级栈空间。

鉴于以上两点原因，这必然涉及到内存分配的工作，咱们的内存管理是在 memory.c 中，代码 11-4 是新增的相关功能。

代码 11-4　（project/c11/b/kernel/memory.c）

```
   …略
24 struct pool {
   …略
28   struct lock lock;           // 申请内存时互斥
29 };
   …略
34 /* 在 pf 表示的虚拟内存池中申请 pg_cnt 个虚拟页,
35  * 成功则返回虚拟页的起始地址，失败则返回 NULL */
36 static void* vaddr_get(enum pool_flags pf, uint32_t pg_cnt) {
37    int vaddr_start = 0, bit_idx_start = -1;
38    uint32_t cnt = 0;
39    if (pf == PF_KERNEL) {  // 内核内存池
    …略
48    } else {         // 用户内存池
49        struct task_struct* cur = running_thread();
50        bit_idx_start  = bitmap_scan(&cur->userprog_vaddr.vaddr_bitmap, pg_cnt);
51        if (bit_idx_start == -1) {
52           return NULL;
53        }
54
55        while(cnt < pg_cnt) {
56           bitmap_set(&cur->userprog_vaddr.vaddr_bitmap, bit_idx_start + cnt++, 1);
57        }
58        vaddr_start = cur->userprog_vaddr.vaddr_start + bit_idx_start * PG_SIZE;
59
60    /* (0xc0000000 - PG_SIZE)作为用户 3 级栈已经在 start_process 被分配 */
61        ASSERT((uint32_t)vaddr_start < (0xc0000000 - PG_SIZE));
62    }
63    return (void*)vaddr_start;
64 }
    …略
176 /* 在用户空间中申请 4k 内存，并返回其虚拟地址 */
177 void* get_user_pages(uint32_t pg_cnt) {
178    lock_acquire(&user_pool.lock);
179    void* vaddr = malloc_page(PF_USER, pg_cnt);
180    memset(vaddr, 0, pg_cnt * PG_SIZE);
181    lock_release(&user_pool.lock);
182    return vaddr;
183 }
184
185 /* 将地址 vaddr 与 pf 池中的物理地址关联，仅支持一页空间分配 */
186 void* get_a_page(enum pool_flags pf, uint32_t vaddr) {
187    struct pool* mem_pool = pf & PF_KERNEL ? &kernel_pool : &user_pool;
188    lock_acquire(&mem_pool->lock);
189
190    /* 先将虚拟地址对应的位图置 1 */
191    struct task_struct* cur = running_thread();
192    int32_t bit_idx = -1;
193
194 /* 若当前是用户进程申请用户内存，就修改用户进程自己的虚拟地址位图 */
195    if (cur->pgdir != NULL && pf == PF_USER) {
196        bit_idx = (vaddr - cur->userprog_vaddr.vaddr_start) / PG_SIZE;
197        ASSERT(bit_idx > 0);
198        bitmap_set(&cur->userprog_vaddr.vaddr_bitmap, bit_idx, 1);
199
200    } else if (cur->pgdir == NULL && pf == PF_KERNEL){
201 /* 如果是内核线程申请内核内存，就修改 kernel_vaddr */
202        bit_idx = (vaddr - kernel_vaddr.vaddr_start) / PG_SIZE;
203        ASSERT(bit_idx > 0);
204         bitmap_set(&kernel_vaddr.vaddr_bitmap, bit_idx, 1);
205    } else {
206         PANIC("get_a_page:not allow kernel alloc userspace or
            user alloc kernelspace by get_a_page");
207   }
208
209    void* page_phyaddr = palloc(mem_pool);
210    if (page_phyaddr == NULL) {
211        return NULL;
212    }
213    page_table_add((void*)vaddr, page_phyaddr);
```

```
214     lock_release(&mem_pool->lock);
215     return (void*)vaddr;
216 }
217
218 /* 得到虚拟地址映射到的物理地址 */
219 uint32_t addr_v2p(uint32_t vaddr) {
220     uint32_t* pte = pte_ptr(vaddr);
221 /* (*pte)的值是页表所在的物理页框地址,
222  * 去掉其低 12 位的页表项属性+虚拟地址 vaddr 的低 12 位 */
223     return ((*pte & 0xfffff000) + (vaddr & 0x00000fff));
224 }
225
226 /* 初始化内存池 */
227 static void mem_pool_init(uint32_t all_mem) {
228     put_str("   mem_pool_init start\n");
        …略
279     lock_init(&kernel_pool.lock);
280     lock_init(&user_pool.lock);
        …略
```

代码 11-4 还是蛮长的，这只是列出了新增的部分，不过新函数都是用旧函数来重新组合实现的，因此需要介绍的不是很多，下面给大伙儿说说。

代码最前部，我们在内存池 struct pool 中新增了锁 struct lock lock，用它来在内存申请时做互斥，避免公共资源的竞争。

在接下来的 vaddr_get 函数中，我们新增了在用户内存池分配内存的功能，即代码第 48～62 行。此部分的处理逻辑同在内核内存池中分配内存一样，大伙儿一看便知。

下一个新增的函数是 get_user_pages，它用来在用户内存池中以整页为单位分配内存，返回分配的虚拟地址，实现较简单，用的都是之前介绍过的函数，无需多余的解释。

另一个新增的函数是 get_a_page，它用来在某个内存池中获取一个页，但与 get_user_pages 和 get_kernel_pages 不同的是此函数原型是：

“get_a_page(enum pool_flags pf, uint32_t vaddr)”

多了个参数 vaddr，vaddr 用来指定绑定的虚拟地址，所以此函数的功能是申请一页内存，并用 vaddr 映射到该页，也就是说我们可以指定虚拟地址。而 get_user_pages 和 get_kernel_pages 不能指定虚拟地址，只能由内存管理模块自动分配虚拟地址，分配什么咱们就用什么。此函数内部实现就是把之前介绍过的方法重新拼合了，都是熟悉的函数，较容易理解。

最后一个要介绍的新函数是 addr_v2p，此函数返回虚拟地址 vaddr 所映射的物理地址。

addr_v2p 的原理是根据页表映射原理，先得到虚拟地址 vaddr 最终所映射到的物理页框起始地址，也就是在页表中 vaddr 所在的 pte 中记录的那个物理页地址，然后再将 vaddr 的低 12 位与此值相加，所得的地址和便是 vaddr 映射的物理地址。这里多说两句，该函数实现中的“uint32_t*pte = pte_ptr(vaddr)”，在指针变量 pte 中得到 vaddr 的所在 pte 的地址，此时*pte 的内容是 vaddr 所在 pte 的内容，也就是 vaddr 最终所映射到的物理页框的 32 位地址中的高 20 位和 12 位的页表项属性，因为页框都是自然页，低 12 位地址是 0，所以页表项 pte（和页目录项 pde）中只需要记录页框的高 20 位地址即可。为了获取 pte 中的地址部分，在此要把低 12 位的属性值去掉，也就是 return 语句中的代码“(*pte & 0xfffff000)”的目的，另外的代码“(vaddr & 0x00000fff)”就是获取原虚拟地址 vaddr 的低 12 位。

最后要说明的是由于我们在内存池 struct pool 中增加了锁，在内存池初始化函数 mem_pool_init 中，我们增加了锁的初始化：“lock_init(&kernel_pool.lock);”和“lock_init(&user_pool.lock);”。

好啦，本节中有关 memory.c 的修改就到这（我隐约听到：“什么，以后还要改?”），说实话，本节虽然增加了不少新东西，但并不是所有介绍的函数在本次中都用到，我还顺便增加了今后要用的一些功能，因为我们还有好多功能没加进去呢，小弟我又不想那么频繁地修改同一文件，若每次只涉及一点点新功能，我是介绍，还是不介绍呢，容易纠结，所以我尽量每次塞进去一些较容易的函数。在不久的将来咱们还要实现更多的功能，到时候那才叫“大修”呢。

11.3.4 进入特权级 3

在有了前面的基础后，终于我们来到了这里，用户进程工作在最低的特权级——特权级 3，可是到目前为止我们都工作在 0 特权级下，如何从特权级 0 迈向特权级 3 呢？

从特权级 0 进入特权级 3 有几个关键点，这主要是涉及特权级方面的内容（忘记特权级是怎么回事的同学赶紧翻回去复习下），下面咱们通过讨论的方式逐步得出结论。

一直以来我们都在 0 特权级下工作，即使是在创建用户进程的过程中也是。可是我们知道，一般情况下，CPU 不允许从高特权级转向低特权级，除非是从中断和调用门返回的情况下。咱们系统中不打算使用调用门，因此，咱们进入特权级 3 只能借助从中断返回的方式，但用户进程还没有运行，何谈被中断？更谈不上从中断返回了……但是 CPU 比较呆头呆脑，我们可以骗过 CPU，在用户进程运行之前，使其以为我们在中断处理环境中，这样便“假装”从中断返回。

如何假装呢？从大体上来看，首先得在特权级 0 的环境中，其次是执行 iretd 指令。这么说太笼统了，下面咱们说得具体点。

从中断返回肯定要用到 iretd 指令，iretd 指令会用到栈中的数据作为返回地址，还会加载栈中 eflags 的值到 eflags 寄存器，如果栈中 cs.rpl 若为更低的特权级，处理器的特权级检查通过后，会将栈中 cs 载入到 CS 寄存器，栈中 ss 载入 SS 寄存器，随后处理器进入低特权级。因此我们必然要在栈中提前准备好数据供 iretd 指令使用。您看，既然已经涉及到栈操作了，不如进行得更彻底一些，咱们将进程的上下文都存到栈中，通过一系列的 pop 操作把用户进程的数据装载到寄存器，最后再通过 iretd 指令退出中断，把退出中断彻底地“假装”一回。现在完全可以再重复写一套退出中断的代码，虽然仅为短短几行，但依然“勤俭节约”，我们可以复用之前的成果，回忆一下，退出中断的出口是汇编语言函数 intr_exit，这是我们定义在 kernel.S 中的，此函数用来恢复中断发生时、被中断的任务的上下文状态，并且退出中断。

由此我们得出关键点 1：从中断返回，必须要经过 intr_exit，即使是“假装”。

在中断发生时，我们在中断入口函数“intr%1entry”中通过一系列的 push 操作来保存任务的上下文，因此在 intr_exit 中恢复任务上下文要通过一系列的 pop 操作，这属于“intr%1entry”的逆过程。

任务的上下文信息被保存在任务 pcb 中的 struct intr_stack 中，注意啦，struct intr_stack 并不要求有固定的位置，它只是一种保存任务上下文的格式结构。

以下为叙述方便，将 struct intr_stack 称为栈。

在汇编函数 intr_exit 里面的一系列的 pop 操作是为了恢复任务的上下文，从它退出中断是不可避免的，这样才能“假装”从中断返回。

由此我们得出关键点 2：必须提前准备好用户进程所用的栈结构，在里面填装好用户进程的上下文信息，借一系列 pop 出栈的机会，将用户进程的上下文信息载入 CPU 的寄存器，为用户进程的运行准备好环境。

当执行完 intr_exit 中的 iretd 指令后，CPU 便恢复了任务中断前的状态：中断前是哪个特权级就进入哪个特权级。

CPU 是如何知道从中断退出后要进入哪个特权级呢？这是由栈中保存的 CS 选择子中的 RPL 决定的，我们知道，CS.RPL 就是 CPU 的 CPL，当执行 iretd 时，在栈中保存的 CS 选择子要被加载到代码段寄存器 CS 中，因此栈中 CS 选择子中的 RPL 便是从中断返回后 CPU 的新的 CPL。

我们进入（假装返回）到 3 特权级，由此我们得出关键点 3：我们要在栈中存储的 CS 选择子，其 RPL 必须为 3。

大伙知道，RPL 是选择子中的低 2 位，用以表示（或者叫“揭露”）访问者特权级，因此 RPL 是为了避免低特权级任务作弊使用指向高特权级内存段的选择子而提供的一种检测手段。虽然 RPL 是 CPU 提供的、硬件级的方案，但 CPU 只负责接收选择子，它自己可不知道所提交的选择子是否是造假的。因此它把 RPL 的维护权交给了操作系统，由操作系统去保证所有提交的选择子都是“真货”。所以，为了避免任务提交一个假的选择子（通常是指向特权级更高的内存段），操作系统会将选择子的 RPL 置为用户进程的 CPL，只有 CPL 和 RPL 在数值上同时小于等于选择子所指向的内存段的 DPL 时，CPU 的安全检测才通过，从而避免了

低特权级任务跨级访问高特权级的内存段。

既然用户进程的特权级为 3，操作系统不能辜负 CPU 的委托，它有责任把用户进程所有段选择子的 RPL 都置为 3，因此，在 RPL=CPL=3 的情况下，用户进程只能访问 DPL 为 3 的内存段，即代码段、数据段、栈段。我们前面的工作中已经准备好了 DPL 为 3 的代码段及数据段，由此我们得出关键点 4，栈中段寄存器的选择子必须指向 DPL 为 3 的内存段。

对于可屏蔽中断来说，任务之所以能进入中断，是因为标志寄存器 eflags 中的 IF 位为 1，退出中断后，还得保持 IF 位为 1，继续响应新的中断。

由此我们得出关键点 5：必须使栈中 eflags 的 IF 位为 1。

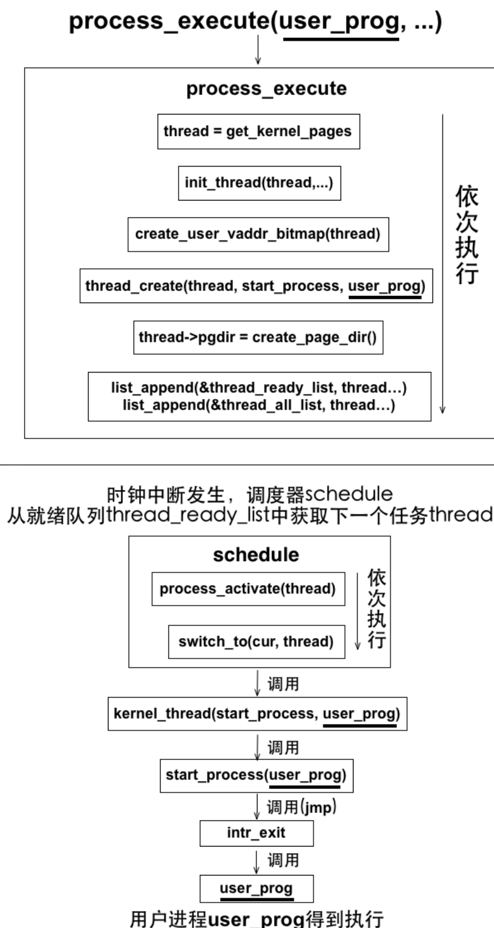
用户进程属于最低的特权级，对于 IO 操作，不允许用户进程直接访问硬件，只允许操作系统有直接的硬件控制。这是由标志寄存器 eflags 中 IOPL 位决定的，必须使其值为 0。

由此我们得出关键点 6：必须使栈中 eflags 的 IOPL 位为 0。

好啦，关键点说完了，下节我们将围绕这 6 点来实现用户进程。

11.3.5　用户进程创建的流程

我们即将介绍用户进程实现的细节。但是如果不从全局上先让大伙儿对整个进程创建的脉络有所掌握的话，今后在介绍局部细节时难免会让大家有摸不着头脑的感觉，最好是先给大伙儿一张“地图”，每走一步您都知道所介绍的代码细节身居何处。为此，为了帮助大伙儿容易地理解进程创建的原理，本节先从全局上做个介绍，进程创建的流程如图 11-12 所示。



▲图 11-12　进程创建流程图

对于图 11-12 中那些未知的函数，此处暂且将疑惑忽略，我们会在本节后介绍，现在的目的是让大伙儿对进程创建的过程有个大致把握。

进程从创建到运行在总体上分为两步，进程创建的工作是由函数 process_execute 完成的，进程的执行是由时钟中断调用 schedule，schedule 从就绪队列中调度进程完成的，毫无疑问的是进程必然是创建在先，执行在后。哈哈，这不是废话吗，其实我有意说明这个顺序只是想强调图 11-12 的阅读顺序是从上到下。

process_execute 的参数是 user_prog，这是待执行的用户进程，在流程图中它出现了 5 次，为了让您了解它是如何被安装的，我在每次它出现的地方加了下画线。由于进程的实现基于线程，故进程创建过程中我们用到了很多创建线程的函数。在 process_execute 中，先调用函数 get_kernel_pages 申请 1 页内存创建进程的 pcb，这里的 pcb 就是 thread，接下来调用函数 init_thread 对 thread 进行初始化。随后调用函数 create_user_vaddr_bitmap 为用户进程创建管理虚拟地址空间的位图。接着调用 thread_create 创建线程，此函数的作用是将函数 start_process 和用户进程 user_prog 作为 kernel_thread 的参数，以使 kernel_thread 能够调用 start_proces(user_prog)。接下来是调用函数 create_page_dir 为进程创建页表，随后通过函数 list_append 将进程 pcb，也就是 thread 加入就绪队列和全部队列，至此用户进程的创建部分完成，现在就等着进程运行了。

进程会在何时运行呢？前面说过了，进程的运行是由时钟中断调用 schedule，由调用器 schedule 调度实现的。当 schedule 从就绪队列中获取的 pcb 恰好是新创建的进程 pcb——thread 时，该进程马上就要被执行了。在 schedule 中，调用了 process_activate 来激活进程或线程的相关资源（页表等），随后通过 switch_to 函数调度进程，根据先前进程创建时函数 thread_create 的工作，已经将 kernel_thread 作为函数 switch_to 的返回地址，即在 switch_to 中退出后，处理器会执行 kernel_thread 函数，"相当于" switch_to 调用 kernel_thread。同样在之前的 thread_create 中，已经将 start_process 和 user_prog 作为了 kernel_thread 的参数，故在 kernel_thread 中可以以此形式调用 start_process(user_prog)。函数 start_process 主要用来构建用户进程的上下文，它会将 user_prog 作为进程"从中断返回"的地址，您懂的，这里的"从中断返回"是假装的，目的是让用户进程顺利进入 3 特权级。由于是从 0 特权级的中断返回，故返回地址 user_prog 被 iretd 指令使用，为了复用中断退出的代码，现在需要跳转到中断出口 intr_exit（kernel.S 中汇编代码完成的函数）处，利用那里的 iretd 指令使返回地址 user_prog 作为 EIP 寄存器的值以使 user_prog 得到执行，故相当于 start_process 调用 intr_exit，intr_exit 调用 user_prog，最终用户进程 user_prog 在 3 特权级下执行。

好啦，以上简单地介绍了进程从创建到执行的原理，下面咱们开始介绍细节，如果在介绍细节过程中"迷失"了，可以再回来看看图 11-12。

11.3.6 实现用户进程——上

本节我们开始正式接触进程创建的工作。

构造用户进程的上下文环境，免不了标志寄存器 eflags 的属性位，这里我们在 global.h 中提前定义了它，大家先看下代码 11-5 大概了解一下。

代码 11-5 （project/c11/b/kernel/global.h）

```
…略
113 #define EFLAGS_MBS     (1 << 1)       // 此项必须要设置
114 #define EFLAGS_IF_1     (1 << 9)      // if 为 1，开中断
115 #define EFLAGS_IF_0     0             // if 为 0，关中断
116 #define EFLAGS_IOPL_3   (3 << 12)
            // IOPL3，用于测试用户程序在非系统调用下进行 IO

117 #define EFLAGS_IOPL_0   (0 << 12)     // IOPL0
118
119 #define NULL ((void*)0)
120 #define DIV_ROUND_UP(X, STEP) ((X + STEP - 1) / (STEP))
121 #define bool int
122 #define true 1
123 #define false 0
124
125 #define PG_SIZE 4096
```

好，定义的属性位不多，大概看下就行了，咱们说正题。

和用户进程相关的文件我们都放在 userprog 目录下，现在创建文件 process.c 来实现用户进程。

直接上菜，请看代码 11-6-1，咱们边看代码边介绍。

代码 11-6-1 （project/c11/b/userprog/process.c）

```
   …略
12 extern void intr_exit(void);
13
14 /* 构建用户进程初始上下文信息 */
15 void start_process(void* filename_) {
16    void* function = filename_;
17    struct task_struct* cur = running_thread();
18    cur->self_kstack += sizeof(struct thread_stack);
19    struct intr_stack* proc_stack = (struct intr_stack*)cur->self_kstack;
20    proc_stack->edi = proc_stack->esi =\
      proc_stack->ebp =proc_stack->esp_dummy = 0;

21    proc_stack->ebx = proc_stack->edx = \
      proc_stack->ecx = proc_stack->eax = 0;
```

```
22    proc_stack->gs = 0;          // 用户态用不上，直接初始为 0
23    proc_stack->ds = proc_stack->es = proc_stack->fs = SELECTOR_U_DATA;
24    proc_stack->eip = function;   // 待执行的用户程序地址
25    proc_stack->cs = SELECTOR_U_CODE;
26    proc_stack->eflags = (EFLAGS_IOPL_0 | EFLAGS_MBS | EFLAGS_IF_1);
27    proc_stack->esp = (void*)((uint32_t)get_a_page(PF_USER,\
      USER_STACK3_VADDR) + PG_SIZE) ;
28    proc_stack->ss = SELECTOR_U_DATA;
29    asm volatile ("movl %0, %%esp; jmp intr_exit" \
      : : "g" (proc_stack) : "memory");
30 }
31
32 /* 激活页表 */
33 void page_dir_activate(struct task_struct* p_thread) {
34 /********************************************************
35  * 执行此函数时，当前任务可能是线程。
36  * 之所以对线程也要重新安装页表，原因是上一次被调度的可能是进程，
37  * 否则不恢复页表的话，线程就会使用进程的页表了。
38  ********************************************************/
39
40 /* 若为内核线程，需要重新填充页表为 0x100000 */
41   uint32_t pagedir_phy_addr = 0x100000;
// 默认为内核的页目录物理地址，也就是内核线程所用的页目录表
42    if (p_thread->pgdir != NULL) {    // 用户态进程有自己的页目录表
43        pagedir_phy_addr = addr_v2p((uint32_t)p_thread->pgdir);
44    }
45
46    /* 更新页目录寄存器 cr3，使新页表生效 */
47    asm volatile ("movl %0, %%cr3" : : "r" (pagedir_phy_addr) : "memory");
48 }
49
50 /* 激活线程或进程的页表，更新 tss 中的 esp0 为进程的特权级 0 的栈 */
51 void process_activate(struct task_struct* p_thread) {
52    ASSERT(p_thread != NULL);
53    /* 激活该进程或线程的页表 */
54    page_dir_activate(p_thread);
55
56    /* 内核线程特权级本身就是 0，处理器进入中断时并不会
      从 tss 中获取 0 特权级栈地址，故不需要更新 esp0*/
57    if (p_thread->pgdir) {
58        /* 更新该进程的 esp0，用于此进程被中断时保留上下文 */
59        update_tss_esp(p_thread);
60    }
61 }
62
```

代码 11-6-1 的开头声明了外部函数 intr_exit，这是用户进程进入 3 特权级的关键。

函数 start_process 接收一个参数 filename_，此参数表示用户程序的名称，用户程序肯定是从文件系统上加载到内存的，因此进程名是进程的文件名。此函数用来创建用户进程 filename_的上下文，也就是填充用户进程的 struct intr_stack，通过假装从中断返回的方式，间接使 filename_运行。

我们说过用户进程是基于线程来实现的，因此在图 11-9 中的线程创建流程中，函数 start_process 相当于最下面的 function，也就是说，我们创建进程的第一步是在线程中运行函数 start_process，后面我们会验证这一点。

下面看下 start_process 的实现，前面的 6 个关键点就是在此函数中体现的，实现部分不长，但要说明的东西有点多，ready？go！

函数体中第一句是“void* function = filename_;”，其实它不重要，但还是解释下吧。

程序最终的舞台是在内存中，CPU 只能直接执行位于内存中的指令。用户进程在执行前，是由操作系统的程序加载器将用户程序从文件系统读到内存，再根据程序文件的格式解析其内容，将程序中的段展开到相应的内存地址。程序格式中会记录程序的入口地址，CPU 把 CS:[E]IP 指向它，该程序就被执行了。C 语言中虽然不能直接控制这两个寄存器，但函数调用其实就是改变这两个寄存器的指向，故 C 语言编写的操作系统可以像调用函数那样调用执行用户程序。因此，用户进程被加载到内存中后同函数一样，仅仅是个指令区域。由于目前我们尚未实现文件系统，前期我们用普通函数代替用户程序，所以用 function 代

替了 filename_，待后面文件系统完成时就可以不用这个蹩脚的名称了。

用户进程上下文保存在 struct intr_stack 栈中，虽然此栈的位置不固定，但我们还得为它安排个合适的位置。

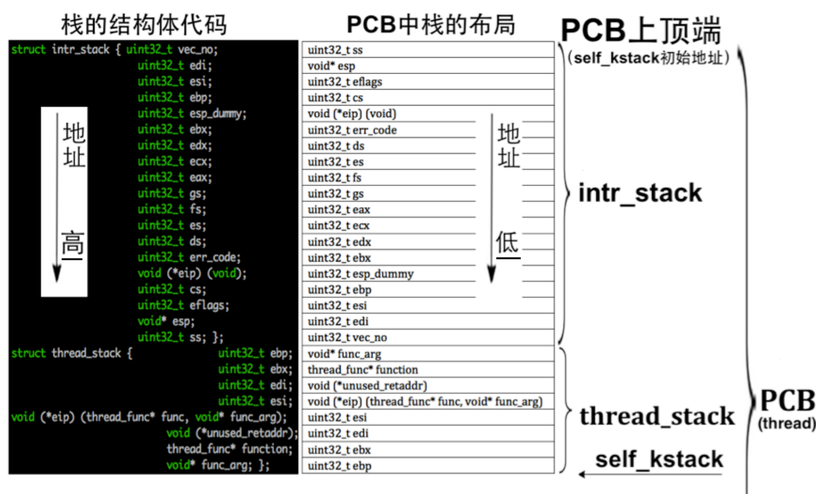
大伙儿一块回忆下创建线程的过程，在函数 init_thread 中有这样一句代码：“pthread->self_kstack = (uint32_t*)((uint32_t)pthread + PG_SIZE);”，目的是初始化线程所用的栈的基址，后面的两个栈“struct intr_stack”和“struct thread_stack”的布局及所占的空间以此基地址往下顺延，这个布局操作是在函数 thread_create 中完成的，相关代码是：

```
“pthread->self_kstack -= sizeof(struct intr_stack);”
```

和

```
“pthread->self_kstack -= sizeof(struct thread_stack);”
```

struct intr_stack 栈用来存储进入中断时任务的上下文，struct thread_stack 用来存储在中断处理程序中、任务切换（switch_to）前后的上下文。这两个栈的布局情况如图 11-13 所示。



▲图 11-13　PCB 中的栈布局_a

在图 11-13 中，左边黑色背景的结构体是实际定义的栈代码，值得注意的是结构体中成员的地址是越往下越高。而右边白色横格是栈，其中的内容是结构体中的成员，地址是越往下越低。

self_kstack 在 init_thread 中被赋予指向 PCB 上顶端，经过上面引号中的两行代码后，self_kstack 指 PCB 中 struct thread_stack 栈的最底端，如图 11-13 中横向箭头的位置所示。

在线程创建过程中，我们把线程的上下文保存在了 struct thread_stack 栈中，实际操作 struct thread_stack 栈的代码如图 11-14 所示。

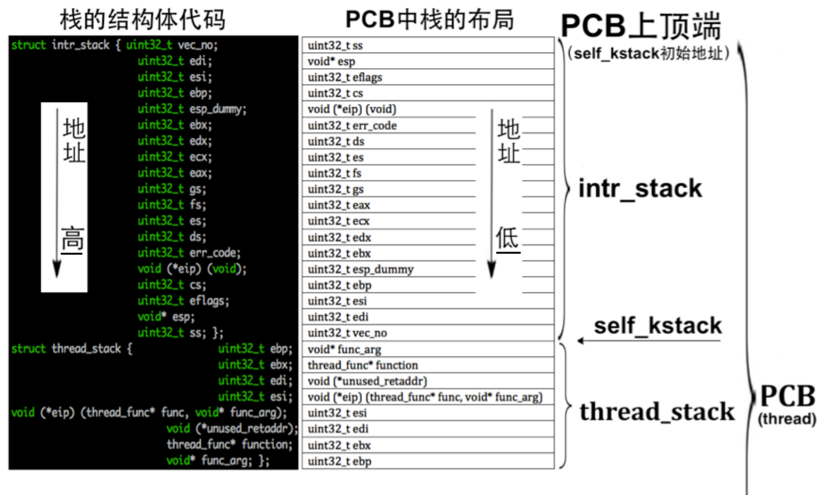
```
33 /* 初始化线程栈thread_stack,将待执行的函数和参数放到thread_stack中相应的位置 */
34 void thread_create(struct task_struct* pthread, thread_func function, void* func_arg) {
35    /* 先预留中断使用栈的空间,可见thread.h中定义的结构 */
36    pthread->self_kstack -= sizeof(struct intr_stack);
37
38    /* 再留出线程栈空间,可见thread.h中定义 */
39    pthread->self_kstack -= sizeof(struct thread_stack);
40    struct thread_stack* kthread_stack = (struct thread_stack*)pthread->self_kstack;
41    kthread_stack->eip = kernel_thread;
42    kthread_stack->function = function;
43    kthread_stack->func_arg = func_arg;
44    kthread_stack->ebp = kthread_stack->ebx = kthread_stack->esi = kthread_stack->edi = 0;
45 }
```

▲图 11-14　线程创建过程

struct thread_stack 栈是由函数 kernel_thread 使用的，之后 kernel_thread 调用 function（最终在线程中执行的函数），function 因此得以执行。

目前栈 struct intr_stack 还是空的，此栈有两个作用，一方面是任务被中断时，用来保存任务的上下文，另一方面这是为了给进程预留的，用来填充用户进程的上下文，也就是寄存器环境。

现在回来看代码 11-6-1，为了引用 struct intr_stack 栈，我们在第 18 行，通过代码“cur->self_ kstack += sizeof(struct thread_stack);”使指针 self_kstack 跨过 struct thread_stack 栈，最终指向 struct intr_stack 栈的最低处，此时 PCB 中栈的情况如图 11-15 所示。



▲图 11-15 PCB 中的栈布局_b

接下来的第 19 行，声明 struct intr_stack* 指针 proc_stack，使其指向 self_kstack，也就是 struct intr_stack 栈的最低处，如图 11-15 中横向箭头指向的位置所示。这么做的原因是结构体指针 proc_stack 指向结构体的最低处，对结构体成员的访问是由低向高处做偏移，这样符合结构体成员访问的方式。

第 20～21 行是对栈中 8 个通用寄存器初始化，在程序开始运行之初，尚未进行任何计算，因此它们都无实际值，把它们初始化为 0 即可。

接下来是对栈中显存段寄存器 gs 初始化，操作系统不允许用户进程访问显存，所以将其初始化为 0。

说明一下，此处不允许用户进程直接控制显存是操作系统的管理方法。其实 CPU 是允许低特权级（用户进程）的任务直接访问显存的，因为显存毕竟是块内存区域，访问内存区域就要通过描述符，因此只要在描述符中把它的 DPL 设置成低特权级就好，也就是显存段的 DPL 数值上大于等于用户特权级即可。不过话又说回来了，即使此处的 gs 不置为 0，CPU 也会将其置 0，原因是执行 iretd 从中断返回时，CPU 会进行特权级检查，如果发现未来的 CPL（也就是内核栈中 CS.RPL）权限低于（数值上大于）CPU 中段寄存器（如 DS、ES、FS、GS）中选择子指向的内存段的 DPL，CPU 会自动将相应段寄存器的选择子置为 0。这样一来，如果低特权级程序用此 0 值选择子访问 GDT，必然会导致访问 GDT 中第 0 个不可访问的哑描述符，导致 CPU 抛异常，从而阻止了越权访问。因此，在特权为 3 的用户环境下 gs 选择子用不上，即使赋值成其他值，由于 cpl 为 3，特权检查时 CPU 就将 gs 置 0 了，干脆这里直接置为 0。

代码第 23 行是将栈中段寄存器 ds、es 和 fs 的值设置为选择子 SELECTOR_U_DATA，此选择子在代码 11-1 的 global.h 中定义。

程序能上 CPU 运行，原因就是 CS:[E]IP 指向了程序入口地址。

第 24 行通过“proc_stack->eip = function;”，先对栈中 eip 赋值为 function，这是 start_process 的参数 filename_的值。

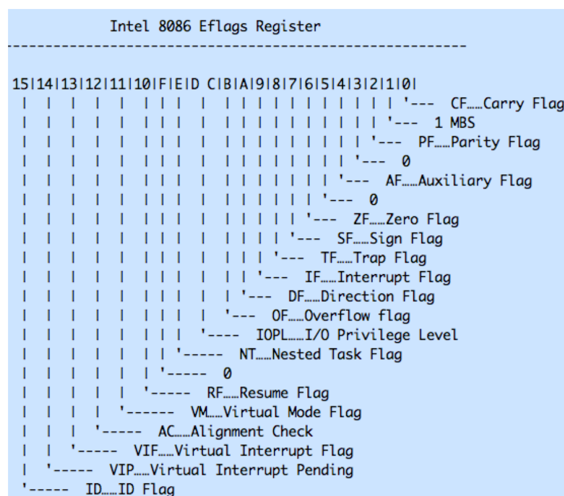
第 25 行通过“proc_stack->cs = SELECTOR_U_CODE”将栈中代码段寄存器 cs 赋值为先前我们已在 GDT 中安装好的用户级代码段。

接下来对栈中 eflags 赋值，EFLAGS_IOPL_0 表示 IOPL 位为 0，EFLAGS_IF_1 表示 IF 位为 1，EFLAGS_MBS 固定为 1，它们在 eflags 中的位置如图 11-16 所示。

接下来第 27 行要为用户进程分配 3 特权级下的栈，也就是栈中 proc_stack->esp 需要指向从用户内存池中分配的地址。

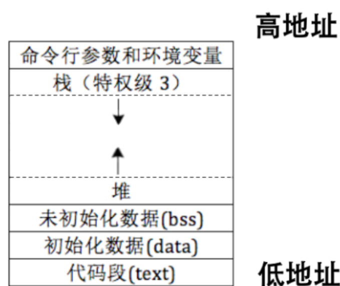
继续之前先介绍下 C 程序的内存分布，如图 11-17 所示。

用户程序内存空间的最顶端用来存储命令行参数及环境变量，这些内容是由某操作系统下的 C 运行库写进去的，将来实现从文件系统加载用户进程并为其传递参数时会介绍这部分。紧接着是栈空间和堆空间，栈向下扩展，堆向上扩展，栈与堆在空间上是相接的，这两个空间由操作系统管理分配，由于栈与堆是相向扩展的，操作系统需要检测栈与堆的碰撞。最下面的未初始化数据段 bss、初始化数据段 data 及代码段 text 由链接器和编译器负责。



▲图 11-16　eflags 中属性位

在 4GB 的虚拟地址空间中，(0xc0000000-1)是用户空间的最高地址，0xc0000000～0xffffffff 是内核空间。



▲图 11-17　C 程序内存布局

我们也效仿这种内存结构布局，把用户空间的最高处即 0xc0000000-1，及以下的部分内存空间用于存储用户进程的命令行参数，之下的空间再作为用户的栈和堆。命令行参数也是被压入用户栈的（在后面章节介绍加载用户进程时会了解），因此虽然命令行参数位于用户空间的最高处，但它们相当于位于栈的最高地址处，所以用户栈的栈底地址为 0xc0000000。由于在申请内存时，内存管理模块返回的地址是内存空间的下边界，所以我们为栈申请的地址应该是（0xc0000000-0x1000），此地址是用户栈空间栈顶的下边界。这里我们用宏来定义此地址，即 USER_STACK3_VADDR，它定义在 userprog.h 中。

“#define USER_STACK3_VADDR　(0xc0000000 - 0x1000)”，

具体代码是用第 27 行的“get_a_page(PF_USER, USER_STACK3_VADDR)”先获取特权级 3 的栈的下边界地址，将此地址再加上 PG_SIZE，所得的和就是栈的上边界，即栈底，将此栈底赋值给 proc_stack->esp。

也许您会有疑问，此处为用户进程创建的 3 特权级栈，它是在谁的内存空间中申请的？换句话说，是安装在谁的页表中了？不会是安装到内核线程使用的页表中了吧？当然不是，用户进程使用的 3 级栈必然要建立在用户进程自己的页表中。您可以回顾一下图 11-12，有两项工作确保了这件事的正确性，在进程创建部分，有一项工作是 create_page_dir，这是提前为用户进入创建了页目录表，在进程执行部分，有一项工作是 process_activate，这是使任务（无论任务是否为新创建的进程或线程，或是老进程、老线程）自己的页表生效。我们是在函数 start_process 中为用户进程创建了 3 特权级栈，start_process 是在执行任务页表激活之后执行的，也就是在 process_activate）之后运行，那时已经把页表更新为用户进程的页表了，所以 3 特权级栈是安装在用户进程自己的页表中的。好啦，暂时解释这么多，如果还是感觉到模糊，建议把此部分了解后，再把代码通读一次就明白了。

栈段可以用普通的数据段，第 28 行为栈中 SS 赋值为用户数据段选择子 SELECTOR_U_DATA。

第 29 行通过内联汇编，将栈 esp 替换为我们刚刚填充好的 proc_stack，然后通过 jmp intr_exit 使程序流程跳转到中断出口地址 intr_exit，通过那里的一系列 pop 指令和 iretd 指令，将 proc_stack 中的数据载入 CPU 的寄存器，从而使程序“假装”退出中断，进入特权级 3。

呼……终于把 start_process 说完了，但心里还有点小事儿要和大家说下：proc_stack 的数据类型是 struct intr_stack，虽然咱们把它定义在 PCB 中，位于 PCB 中最顶端，但它完全可以用局部变量代替，因为它只用这

一次，之后不需要再次访问。故可以在函数 start_process 中这样声明 proc_stack："struct intr_stack proc_stack"，后面的填充操作完全一样，而且也用不着执行第 17～18 行的代码了。不过，既然此时 PCB 顶端的 struct intr_stack 空间还空着，不用就浪费了，哈哈，那咱们就用了它吧。

start_process 函数的介绍到此为止，在介绍下一个函数之前，咱们还要"跑偏"一次。

我们知道，每个进程都拥有独立的虚拟地址空间，本质上就是各个进程都有单独的页表，页表是存储在页表寄存器 CR3 中的，CR3 寄存器只有 1 个，因此，不同的进程在执行前，我们要在 CR3 寄存器中为其换上与之配套的页表，从而实现了虚拟地址空间的隔离。

好啦，回到正题。

下面要介绍的函数是 page_dir_activate，它接受一个参数 p_thread，用来激活 p_thread 的页表，p_thread 可能是进程，也可能是线程。

也许您会有疑问，进程才有独立的地址空间，才有自己的页表，按理说激活页表这类工作应该针对的是进程啊，为什么线程也需要呢？

在第 34～38 行的注释解释了这一点，目前咱们的线程并不是为用户进程服务的，它是为内核服务的，因此与内核共享同一地址空间，也就是和内核用的是同一套页表。当进程 A 切换到进程 B 时，页表也要随之切换到进程 B 所用的页表，这样才保证了地址空间的独立性。当进程 B 又切换到线程 C 时，由于目前在页表寄存器 CR3 中的还是进程 B 的页表，因此，必须要将页表更换为内核所使用的页表。所以，无论是针对进程，还是线程，都要考虑页表切换。

第 41 行，将 pagedir_phy_addr 初始化为 0x100000，这是内核所使用的页表的物理地址，也是所有内核线程的页表。

如何判断当前任务是线程，还是进程呢？老办法，还是判断 pcb 中的 pgdir 是否等于 NULL，pcb->pgdir 用来指向页表的虚拟地址。线程 pcb 中是没有页表的，所以 pgdir 等于 NULL，如果是进程，其 pgdir 不为 NULL。

按照这个思路，在第 42 行，通过代码"if (p_thread->pgdir != NULL)"判断是否为进程，若为进程，则将进程的页表地址加载到 CR3 寄存器。

注意，pgdir 中的是页表的虚拟地址，因为页表需要单独的内存空间，创建页表（由代码 11-6-2 中的函数 create_page_dir 完成）时必然要为页表申请内存，内存管理模块返回的地址是虚拟地址，因此页表地址也是虚拟地址，所以在把页表加载到 CR3 之前，咱们要将其转换成物理地址。

这是第 43 行的"pagedir_phy_addr = addr_v2p((uint32_t)p_thread->pgdir)"完成的，通过函数 addr_v2p 将虚拟地址 p_thread->pgdir 转换为物理地址，重新为变量 pagedir_phy_addr 赋值。

接着在第 47 行，通过内联汇编，将 pagedir_phy_addr 的值通过 mov 指令写入到寄存器 CR3 中，由此实现了页表切换。

本节最后要介绍的函数是 process_activate，它的功能有两个，一是激活线程或进程的页表，二是更新 tss 中的 esp0 为进程的特权级 0 的栈。大伙儿知道，进程与线程都是独立的执行流，它们有各自的栈和页表，只不过线程的页表是和其他线程共用的，而进程的页表是单独的。进程或线程在被中断信号打断时，处理器会进入 0 特权级，并会在 0 特权级栈中保存进程或线程的上下文环境。如果当前被中断的是 3 特权级的用户进程，处理器会自动到 tss 中获取 esp0 的值作为用户进程在内核态（0 特权级）的栈地址，如果被中断的是 0 特权级的内核线程，由于内核线程已经是 0 特权级，进入中断后不涉及特权级的改变，所以处理器并不会到 tss 中获取 esp0，言外之意是即使咱们更新了线程 tss 中的 esp0，处理器也不会使用它，咱们就白费劲了。所以，在代码 11-6-1 的第 57 行用这句代码"if (p_thread->pgdir)"来判断：如果是用户进程的话才去更新 tss 中的 esp0。这两个功能的实现，就是调用 tss.c 中定义的 update_tss_esp 和上面刚介绍的 page_dir_activate 完成的。顺便说一下，只有在任务调度时才会切换页表及更新 0 级栈，因此 process_activate 是被 schedule 调用的，后面咱们会梳理用户进程创建的流程。

由于和用户进程相关的内容有点多，本节就先到这啦，前面还有一段路要走。

11.3.7 bss 简介

我们要实现用户进程的堆内存管理，现在就要着手规划用户进程内存空间布局的基础工作了。在继续下一步之前，咱们要先介绍点 bss 的内容，它涉及到咱们堆内存管理的实现。

用户进程的内存空间布局我们也参照着 Linux 下 C 程序的布局方案来做，也就是图 11-15，麻烦大伙儿再回顾下该图，在 C 程序的内存空间中，位于低处的三个段是代码段、数据段和 bss 段，它们由编译器和链接器规划地址空间，在程序被操作系统加载之前它们地址就固定了。而堆是位于 bss 段的上面，栈是位于堆的上面，它们共享 4GB 空间中除了代码段、数据段及顶端命令行参数和环境变量等以外的其余可用空间，它们的地址由操作系统来管理，在程序加载时为用户进程分配栈空间，运行过程中为进程从堆中分配内存。堆向上扩展，栈向下扩展，因此在程序的加载之初，操作系统必须为堆和栈分别指定起始地址。

在 Linux 中，堆的起始地址是固定的，这是由 struct mm_struct 结构中的 start_brk 来指定的，堆的结束地址并不固定，这取决于堆中内存的分配情况，堆的上边界是由同结构中的 brk 来标记的。

堆是要安排在 bss 以上的，按理说我们只要找到 bss 的结束地址就可以自由规划堆的起始地址了。但其实我们不需要这么麻烦，有更简单省事的做法，为解释清楚这个问题，这里对 bss 多说两句。

大伙儿知道，C 程序大体上分为预处理、编译、汇编和链接四个阶段。根据语法规则，编译器会在汇编阶段将汇编程序源码中的关键字 section 或 segment（汇编源码中，通常用语法关键字 section 或 segment 来逻辑地划分程序区域，虽然很多书中都称此程序区域为段，但实际上它们就是节 section。注意啦，这里所说的 segment 和 section 是指汇编语法中的关键字，仅指它们在汇编代码中的语法意义相同，并不是指编译、链接中的 section 和 segment）编译成节，也就是之前介绍的 section，此时只是生成了目标文件，目标文件中的这些节还不是程序空间中的独立的代码段或数据段，或者说仅仅是代码段或数据段的一部分。链接器将这些目标文件中属性相同的节（section）合并成段（segment），因此一个段是由多个节组成的，我们平时所说的 C 程序内存空间中的数据段、代码段就是指合并后的 segment。

为什么要将 section 合并成 segment？这么做的原因也很简单，一是为了保护模式下的安全检查，二是为了操作系统在加载程序时省事。

大伙儿已经知道，在保护模式下对内存的访问必须要经过段描述符，段描述符用来描述一段内存区域的访问属性，其中的 S 位和 TYPE 位可组合成多种权限属性，处理器用这些属性来限制程序对内存的使用，如果程序对某片内存的访问方式不符合该内存所对应的段描述符（由访问内存时所使用的选择子决定）中设置的权限，比如对代码这种具备只读属性的内存区域执行了写操作，处理器会检查到这种情况并抛出 GP 异常。程序必须要加载到内存中才能执行，为了符合安全检查，程序中不同属性的节必须要放置到合适的段描述符指向的内存中。比如为程序中具有只读可执行的指令部分所分配的内存，最好是通过具有只读、可执行属性的段描述符来访问，否则若通过具有可写属性的段描述符来访问指令区域的话，程序有可能会将自己的指令部分改写，从而引起破坏。处理器对内存访问的安全检查主要体现在使用的段描述符，段描述符是由选择子决定的，而选择子是由操作系统提供的，所以针对程序中不同属性的区域，操作系统得知道用哪个段描述符来匹配程序中这些不同属性的区域片段，也就是要在程序运行之前提前设置程序在运行时各种段寄存器（如 cs、ds）中的选择子。

综上所述，在操作系统的视角中，它只关心程序中这些节的属性是什么，以便加载程序时为其分配不同的段选择子，从而使程序内存指向不同的段描述符，起到保护内存的作用。因此最好是链接器把目标文件中属性相同的节合并到一起，这样操作系统便可统一为其分配内存了。按照属性来划分节，大致上有三种类型。

（1）可读写的数据，如数据节.data 和未初始化节.bss。

（2）只读可执行的代码，如代码节.text 和初始化代码节.init。

（3）只读数据，如只读数据节.rodata，一般情况下字符串就存储在此节。

经过这样的划分，所有节都可归并到以上三种之一，这样方便了操作系统加载程序时的内存分配。由链接器把目标文件中相同属性的节归并之后的节的集合，便称为 segment，它存在于二进制可执行文件中，也就是 C 程序运行时内存空间中分布的代码段、数据段等段。

现在可以正式说说 bss 了，我们已经知道，text 段是代码段，里面存放的是程序的指令，data 段是数据段，里面存放的是程序运行时的数据，它们的共同点是都存在于程序文件中，也就是在文件系统中存在，原因很简单，它们是程序运行必备的“材料”，必须提前准备好，基本上是固定好的内容。而 bss 并不存在于程序文件中，它仅存在于内存中，其实际内容是在程序运行过程中才产生的，起初并无意义，换句话说，程序在运行时的那一瞬间并不需要 bss，因此完全不需要事先在程序文件中存在，程序文件中仅在 elf 头中有 bss 节的虚拟地址、大小等相关记录，这通常是由链接器来处理的，对程序运行并不重要，因此程序文件中并不存在 bss 实体。bss 中的数据是未初始化的全局变量和局部静态变量，程序运行后才会为它们赋值，因此在程序运行之初，里面的数据没意义，由操作系统的程序加载器将其置为 0 就可以了，虽然这些未初始化的全局变量及局部静态变量起初是用不上的，但它们毕竟也是变量，即使是短暂的生存周期也要占用内存，必须提前为它们在内存中“占好座”，bss 区域的目的也正在于此，就是提前为这些未初始化数据预留内存空间。

小总结一下：未运行之前或运行之初，程序中 bss 中的内容都是未初始化的数据，它们也是变量，只不过这些变量的值在最初时是多少都无所谓，它们的意义是在运行过程中才产生的，故程序文件中无需存在 bss 实体，因此不占用文件大小。在程序运行后那些位于 bss 中的未初始化数据便被赋予了有意义的值，那时 bss 开始变得有意义，故 bss 仅存在于内存中。您看，既然 bss 中的数据也是变量，就肯定要占用内存空间，需要把空间预留出来，但它们并不在文件中存在，对于这种只占内存又不占文件系统空间的数据，链接器采取了合理的做法：由于 bss 中的内容是变量，其属性为可读写，这和数据段属性一致，故链接器将 bss 占用的内存空间大小合并到数据段占用的内存中，这样便在数据段中预留出 bss 的空间以供程序在将来运行时使用。注意，这里所说的是 bss 的尺寸会被合并到数据段，并不是 bss 中的实际内容也会被合并到数据段中，毕竟起初 bss 中的内容无意义，将它的内容合并到其他段中真的是“毫无意义”。当程序文件被操作系统加载器加载时，加载器会为程序的各个段分配内存，由于 bss 已被归并到数据段中，故 bss 仅存在于数据段所在的内存中。因此，bss 的作用就是为程序运行过程中使用的未初始化数据变量提前预留了内存空间。程序的 bss 段（数据段的一部分）会由该加载器填充为 0。由此可见，为生成在某操作系统下运行的用户程序，编译器和操作系统需要相互配合。

下面看下 bss 在实际可执行文件中合并的情况，我们拿 find 命令来举例，如图 11-18 所示。

```
[work@localhost ~]$ readelf -e /bin/find
  ...略
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
   ...略
  [26] .data             PROGBITS        0806f320 028320 000314 00  WA  0   0 32
  [27] .dynbss           PROGBITS        0806f640 028640 000044 00  WA  0   0 32
  [28] .bss              NOBITS          0806f684 028684 000dc0 00  WA  0   0 32
  ...略
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08047034 0x08047034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08047134 0x08047134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08047000 0x08047000 0x27d24 0x27d24 R E 0x1000
  LOAD           0x028000 0x0806f000 0x0806f000 0x00684 0x01444 RW  0x1000
  DYNAMIC        0x028014 0x0806f014 0x0806f014 0x000e0 0x000e0 RW  0x4
  NOTE           0x000148 0x08047148 0x08047148 0x00044 0x00044 R   0x4
  GNU_EH_FRAME   0x0227f8 0x080697f8 0x080697f8 0x00d24 0x00d24 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .dynstr .gnu.liblist .gnu.conflict
 .gnu.hash .dynsym .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini
 .rodata .eh_frame_hdr .eh_frame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .dynbss .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
[work@localhost ~]$ sh ~/tool/calculator.sh 0xdc0+0x00684 x
1444
```

▲图 11-18　find 命令的 elf 头

这里用 readelf –e /bin/find 查看 find 程序的 elf 头，下面的是 readelf 的输出。在“Section Headers”中是 find 程序全部节的信息，索引为 28 的节是.bss，我们在图中已经用下画线标出来了，它的 Size 为 0x000dc0，表示 bss 节占用的大小。在“Program Headers”中是 find 程序全部段的信息，第 0 个段是 PHDR 段，它的类型是程序头表，表示程序头表在文件或内存中的位置及大小，第 1 个段是 INTERP 段，它是个字符串，用来指向 ELF 解析器的路径，咱们要关注下面的两个类型为 LOAD 的段，它表示这是一个可装载的段，操作系统的程序加载器负责把此类型的段拷贝到此段规定的内存地址处，也就是 VirtAddr 中记录的地址，用户程序的装载本质上就是将用户程序中类型为 LOAD 的段拷贝到指定的内存地址。

在图中下半部分是“Section to Segment mapping:”，这是链接器将节合并为段的结果展示，也就是哪些节合并到哪个段中。我们看索引为 02 的段，右边有三行的节合并到此段中，其中包括了.text 节，这是标准的代码节，这在一定程度上说明此段很可能是代码段，为了验证，我们可以用此索引 02 在“Program Headers”中查看第 02 个段，该段是类型为 LOAD 的段，其 Flg 标志中是“R E”，这表示可读可执行，充分说明了代码节.text 确实被合并成了代码段，这里已经用线把它们的对应关系连接起来了。“Section to Segment mapping:”中索引为 03 的段，其包括一行的节，里面有数据节.data，这说明 03 段很可能是数据段，此段中还包括了.bss 节，这说明链接器将.bss 节、.data 节等合并到了同一个段中，该段很可能是数据段，为了验证是否为数据段，同样用此索引 03 到“Program Headers”中查看第 03 个段，该段是第二个 LOAD 段，其 Flg 为“RW”，即“可读写”，充分说明此段是数据段，.text 和.data 等节的合并关系也用线标出了。

通常情况下，段在文件中的大小 FileSiz 和在内存中的大小 MemSiz 应该是一致的，而且在任何情况下，段的 MemSiz 都不会比 FileSiz 小。如果在某段中合并了 bss 节，该段的 MemSiz 应该大于 FileSiz，原因是 bss 节不占用文件系统空间，只占用内存空间。根据此思路，我们顺便在数值上也验证下.bss 节是否被合并到了数据段中，您看，数据段的 FileSiz 和 MemSiz 不一致，分别是 0x00684 和 0x01444。这两个数值的差恰好是.bss 节的大小，在图中的最下行，这里是用.bss 节内存大小 0xdc0+数据段的文件大小 0x00684 来验证的，其和为数据段的内存大小 0x1444，与我们的预想吻合，充分证明 bss 合并到数据段中了。好啦，有关 bss 的内容就到此为止。

说了这么多 bss 的内容，您不禁要问了，我这是要干吗？咳咳，还不是为了要介绍堆的实现嘛。在 C 语言中，函数 malloc 用来动态申请内存，所谓的动态内存申请，是指程序在运行中申请的内存，并不是在程序加载时由操作系统加载器为程序段分配的“固定、静态”内存，这种动态申请的内存就是操作系统从申请者自己的堆中分配的。

我们在不久的将来也要实现 malloc，因此也必须支持堆内存管理，前面说过了，堆的起始地址应该在 bss 之上，现在我们知道 bss 已经被归并到数据段了，数据段的类型是可加载的 LOAD 型，程序将来加载运行时，操作系统的程序加载器会为该程序的数据段分配内存，也就是 bss 段的内存区域也会顺便被分配，因此我们不需要单独知道 bss 的结束地址，只要知道数据段的起始地址及大小，便可以确定堆的起始地址了。有关程序加载的内容，将来我们实现了从文件系统中加载运行用户程序时大伙儿就清楚了，目前我们只要了解虽然堆的起始地址应该在 bss 之上，但由于 bss 已融合到数据段中，要实现用户进程的堆，已不需要知道 bss 的结束地址，将来咱们加载程序时会获取程序段的起始地址及大小，因此只要堆的起始地址在用户进程地址最高的段之上就可以了。

好啦，本节到此结束，下一节咱们继续完成用户进程。

11.3.8 实现用户进程——下

上回说到 process.c 的上半场，下面继续介绍实现用户进程的其他函数，大伙儿请看代码 11-6-2。

代码 11-6-2 （project/c11/b/userprog/process.c）

```
   …略
63 /* 创建页目录表，将当前页表的表示内核空间的 pde 复制，
64  * 成功则返回页目录的虚拟地址，否则返回-1 */
65 uint32_t* create_page_dir(void) {
66
67    /* 用户进程的页表不能让用户直接访问到，所以在内核空间来申请 */
68    uint32_t* page_dir_vaddr = get_kernel_pages(1);
```

```
 69    if (page_dir_vaddr == NULL) {
 70        console_put_str("create_page_dir: get_kernel_page failed!");
 71        return NULL;
 72    }
 73
 74 /************************* 1  先复制页表  ***************************/
 75  /* page_dir_vaddr + 0x300*4 是内核页目录的第 768 项 */
 76    memcpy((uint32_t*)((uint32_t)page_dir_vaddr + 0x300*4),\
            (uint32_t*)(0xfffff000+0x300*4), \
            1024);
 77 /*****************************************************************/
 78
 79 /********************* 2  更新页目录地址 ********************/
 80    uint32_t new_page_dir_phy_addr = addr_v2p((uint32_t)page_dir_vaddr);
 81    /* 页目录地址是存入在页目录的最后一项，
       更新页目录地址为新页目录的物理地址 */
 82    page_dir_vaddr[1023] = new_page_dir_phy_addr\
 | PG_US_U | PG_RW_W | PG_P_1;
 83 /*************************************************************/
 84    return page_dir_vaddr;
 85 }
 86
 87 /* 创建用户进程虚拟地址位图 */
 88 void create_user_vaddr_bitmap(struct task_struct* user_prog) {
 89    user_prog->userprog_vaddr.vaddr_start = USER_VADDR_START;
 90    uint32_t bitmap_pg_cnt = DIV_ROUND_UP(\
       (0xc0000000 - USER_VADDR_START) / PG_SIZE / 8 , \
       PG_SIZE);
 91    user_prog->userprog_vaddr.vaddr_bitmap.bits =\
       get_kernel_pages(bitmap_pg_cnt);
 92    user_prog->userprog_vaddr.vaddr_bitmap.btmp_bytes_len =\
       (0xc0000000 - USER_VADDR_START) / PG_SIZE / 8;
 93    bitmap_init(&user_prog->userprog_vaddr.vaddr_bitmap);
 94 }
 95
 96    /* 创建用户进程 */
 97    void process_execute(void* filename, char* name) {
 98      /* pcb 内核的数据结构，由内核来维护进程信息，
       因此要在内核内存池中申请 */
 99    struct task_struct* thread = get_kernel_pages(1);
100    init_thread(thread, name, default_prio);
101    create_user_vaddr_bitmap(thread);
102    thread_create(thread, start_process, filename);
103    thread->pgdir = create_page_dir();
104
105    enum intr_status old_status = intr_disable();
106    ASSERT(!elem_find(&thread_ready_list, &thread->general_tag));
107    list_append(&thread_ready_list, &thread->general_tag);
108
109    ASSERT(!elem_find(&thread_all_list, &thread->all_list_tag));
110    list_append(&thread_all_list, &thread->all_list_tag);
111    intr_set_status(old_status);
112 }
```

在代码 11-6-2 的开头是函数 create_page_dir，此函数用来创建页表，确切地说是创建页目录表。

大伙儿知道，操作系统是为用户进程服务的，它提供了各种各样的系统功能供用户进程调用。为了用户进程可以访问到内核服务，必须确保用户进程必须在自己的地址空间中能够访问到内核才行，也就是说内核空间必须是用户空间的一部分，咱们看看是如何做到这一点的。

虚拟地址空间由页表来控制，页表由操作系统管理，因此用户进程的虚拟空间是由操作系统规划分配的。

每个用户进程都拥有 4GB 虚拟地址空间，操作系统把这 4GB 空间分为用户空间和内核空间两部分，因此内核空间和用户空间的大小是不固定的，它们可以以任意比例分配这 4GB，比如内核和用户空间可以各占 2GB。

Linux 分了 3GB 给用户空间，自己本身占 1GB，所有用户进程的最高 1GB 空间都指向 Linux 所在的内存空间，这样操作系统就被所有用户进程共享。这种共享操作系统的内存布局如图 11-19 所示。

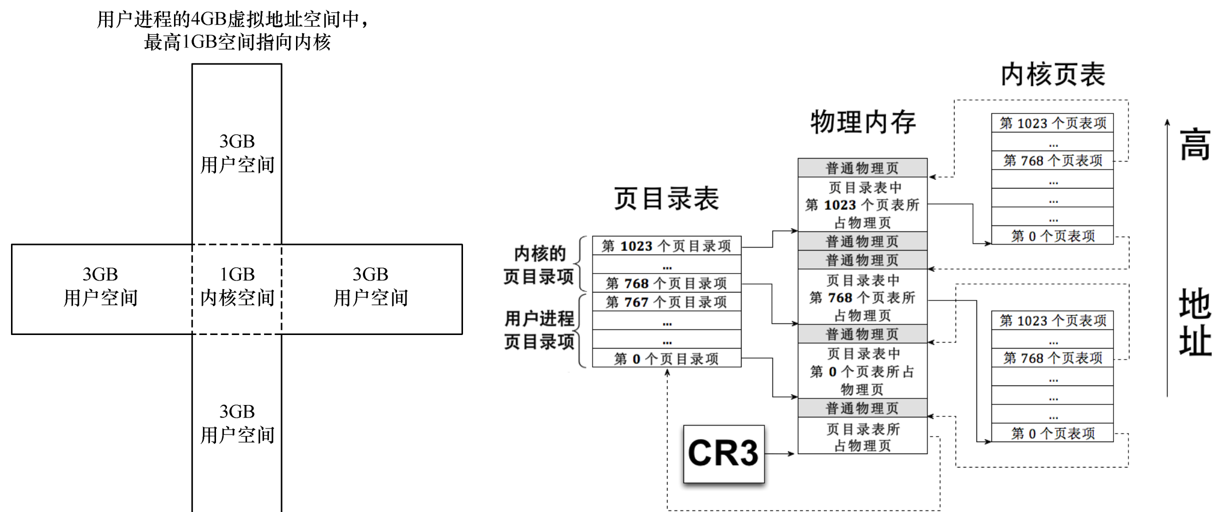
让操作系统被所有用户进程共享，这是如何做到的呢？

我想您可能想到了，既然用户空间是由页表来表示的，那么必须通过设置页表来解决。

让我们复习下页表的知识，我们目前使用的是二级页表，加载到页目录表寄存器 CR3 中的是页目录表的物理地址，页目录表中一共包含 1024 个页目录项（pde），页目录项大小为 4B，故页目录表大小为 4KB。每个页目录项仅表示 1 个页表，页目录项中存储的是页表所在物理页框的物理地址及页目录项的属性。每个页表可容纳 1024 个页表项（pte），页表项大小为 4B，故每个页表本身占用 4KB。每个页表项仅表示一个物理页框，页表项中存储的是 4KB 大小的物理页框的物理地址及页表项的属性，因此每个页表可表示的地址空间为 1024*4KB=4MB，一个页目录表中可包含 1024 个页表，因此可表示 1024*4MB=4GB 大小的地址空间。

目前我们的内核位于 0xc0000000 以上的地址空间，也就是位于页目录表中第 768～1023 个页目录项所指向的页表中，这一共是 256 个页目录项，即 1GB 空间（当然我们的内核没那么大，不会把 1GB 空间占满），目前页表与内核的关系如图 11-20 所示。

图 11-20 是任意进程的页目录表，其中，用户进程占据页目录表中第 0～767 个页目录项，内核占据页目录表中第 768～1023 个页目录项。



▲图 11-19 操作系统被所有用户进程共享　　▲图 11-20 内核页目录表映射

由于第 0～767 个页目录项属于用户空间，图 11-20 只想表达内核所用的页表，故在中间的物理内存中，“内核页目录表中第 0 个页表所占物理页”未画出相关页表。

大伙儿知道，即使是同一个用户程序在被加载多次时，操作系统每次为它们分配的物理内存也不会固定，但操作系统不一样。

首先，操作系统启动后，其在物理内存中的位置是固定的，不会再变动。

其次，操作系统只有一套页表，它们也是固定的。

我们的目的是只要在用户进程中能够访问到内核即可，现在有两个方法。

（1）为每一个用户进程单独准备一份内核的拷贝映像。

（2）为每一个用户进程准备一份内核的符号链接（软链接）。

第 1 个办法不太理想，进程越多，内核拷贝越多，这受限于物理内存大小及实际内核大小，虽然可以先将数据缓存到硬盘，然后借助 pagefault 异常将数据调回内存，但这种做法效率太低了，咱们看看第 2 个方法，为内核准备“符号链接”。

来个小插曲，什么是符号链接？

符号链接是 Linux 系统中的概念。文件都有名称，对于用户来说，文件是通过文件名来访问的，可以把文件名理解为存储在磁盘上的文件实体的访问入口。符号链接是为同一个文件实体多创建了一个访问入口，相当于为原文件起个别名，就像人有大名和小名，都是指同一个人。咱们拿图 11-21 来说明符号链接。

如图所示，原文件 obj 是空的，用 ln 命令为 obj 文件创建了 2 个符号链接，分别名为“sl1_to_obj”和“sl2_to_obj”，也就是说，对于文件 obj 在磁盘上的文件实体来说，现在除了文件名 obj 外，还可以通过

“sl1_to_obj”和“sl2_to_obj”访问它。接下来分别通过文件名obj、“sl1_to_obj”和“sl2_to_obj”写字符“abc”、“xyz”和“123”，通过 cat 命令显示文件内容，发现都是写进了同一个文件。因此，符号链接的作用就是多了个“访问入口”。

小插曲插播结束，赶紧回来。

页表是记录在页目录项中，此处页目录项对于内核物理内存的作用，相当于 Linux 中文件的符号链接，页目录项是访问内核所在物理内存的入口。

因此，为了访问到内核，我们只要给每个用户进程创建访问内核的入口即可。

也就是把用户进程页目录表中的第 768～1023 个页目录项用内核页目录表的第 768～1023 个页目录项代替，其实就是将内核所在的页目录项复制到进程页目录表中同等位置，这样就能让用户进程的高 1GB 空间指向内核。

每创建一个新的用户进程，就将内核页目录项复制到用户进程的页目录表，这样就为内核物理内存创建了多个入口，从而实现了图 11-19 中的所有用户进程共享内核。

```
[work@localhost tmp]$ touch obj
[work@localhost tmp]$ cat obj
[work@localhost tmp]$ ln -s obj sl1_to_obj
[work@localhost tmp]$ ln -s obj sl2_to_obj
[work@localhost tmp]$ ll
总计 0
-rw-rw-r-- 1 work work 0 03-20 14:50 obj
lrwxrwxrwx 1 work work 3 03-20 14:51 sl1_to_obj -> obj
lrwxrwxrwx 1 work work 3 03-20 14:51 sl2_to_obj -> obj
[work@localhost tmp]$ echo 'abc' >> obj
[work@localhost tmp]$ cat obj
abc
[work@localhost tmp]$ echo 'xyz' >> sl1_to_obj
[work@localhost tmp]$ cat obj
abc
xyz
[work@localhost tmp]$ cat sl1_to_obj
abc
xyz
[work@localhost tmp]$ echo '123' >> sl2_to_obj
[work@localhost tmp]$ cat obj
abc
xyz
123
[work@localhost tmp]$ cat sl1_to_obj
abc
xyz
123
[work@localhost tmp]$ cat sl2_to_obj
abc
xyz
123
[work@localhost tmp]$
```

▲图 11-21　Linux 中符号链接

好，咱们已经讨论清楚了进程共享内核的原理，函数 create_page_dir 正是按照此原理实现的，咱们现在说代码。

第 68 行，通过 get_kernel_pages(1)在内核内存池中申请一页内存，将返回地址存储到指针变量 page_dir_vaddr 中，注意，page_dir_vaddr 中的值是虚拟地址。

接下来的工作分为两部分，首先我们要把内核的页目录项复制到用户进程使用的页目录表中。这是通过第 76 行的 memcpy 完成的，咱们分析下这句代码。

memcpy 的第一个形参是复制的目标地址，此处的实参为(uint32_t*)((uint32_t)page_dir_vaddr + 0x300*4)，其中 page_dir_vaddr 是为用户进程申请作为页目录表的基地址，0x300 是十进制的 768，4 是每个页目录项的大小，0x300*4 表示 768 个页目录项的偏移量，因此，这表示复制的目标地址为偏移用户进程页目录表基地址 768 个页目录项的地方，此处正是内核起始地址 0xc0000000 被映射的页表所在的页目录项地址。

memcpy 的第二个形参是复制的源地址，此处的实参是(uint32_t*)(0xfffff000+0x300*4)，用户进程的创建是在内核中完成的，因此目前是在内核的页表中，其中 0xfffff000 便是用来访问内核页目录表的基地址（也是第 0 个页目录项），0x300*4 是内核起始页目录项在页目录表中的偏移量，也就是内核起始地址 0xc0000000 所在的页目录项地址，因此 0xfffff000+0x300*4 是内核页目录表中第 768 个页目录项的地址。

memcpy 的第三个形参是复制的字节量，这里是 1024，即 1024/4=256 个页目录项的大小。

这样内核占用的页目录项被复制到了用户进程的页目录表中，也就是为用户进程创建了访问内核的入口。

其次需要把用户页目录表中最后一个页目录项更新为用户进程自己的页目录表的物理地址。

这么做的原因是将来用户进程运行时，执行期间有可能会有页表操作，页表操作是由内核代码完成的，因此内核需要知道该用户进程的页目录表在哪里。

例如，如果用户进程通过系统调用申请内存，它会陷入内核态，那时内核除了为其分配内存外，如果申请的内存大小跨越了物理页框（大于 4KB），甚至跨越了页表（大于 4MB），还需要在它的页目录表中创建页目录项和在页表中创建页表项，否则会引起 pagefault 异常。每个用户有自己单独的页表，为了让用户进程能够使用系统为其分配的地址空间，肯定需要内核事先在该用户进程自己的页表中创建该地址对应的页目录项和页表项，无论怎样操作页表，必须要让内核获取页目录表的地址。大伙儿已经知道，内核访问页目录表的方法是通过虚拟地址 0xfffff000，这会访问到当前页目录表的最后一个页目录项。为了保证内核操作的是该用户进程自己的页目录表，此时必须把页目录表的物理地址写入用户进程页目录表的最后一个页目录项中。

您看，这里说的是页目录表的物理地址，我们通过 get_kernel_pages 返回的 page_dir_vaddr 是虚拟地址，

因此必须将其转换为物理地址。这是在第 80 行用函数 addr_v2p((uint32_t)page_dir_vaddr)完成的，转换后的地址存储到变量 new_page_dir_phy_addr 中。

第 81 行，将物理地址 new_page_dir_phy_addr 加上属性 PG_US_U 和 PG_RW_W 以及 PG_P_1 后，写入用户进程页目录表的最后一个页目录项，即 page_dir_vaddr[1023]。

至此，create_page_dir 函数完成，通过 return page_dir_vaddr 返回页表的虚拟地址。

我们知道，用户进程有自己的 4GB 虚拟地址空间，这空间中除了存放用户进程自己的指令和数据外，还要包括用户进程自己的堆和栈，用户进程可以在自己的堆中申请、释放内存，因此必须有一套方法跟踪内存的分配情况。和内核一样，用户进程也是用位图来管理地址分配的，每个进程有自己单独的位图，存储在进程 pcb 中的 userprog_vaddr 中。回忆一下，在 C 语言中用户进程用 malloc 申请的内存是在进程自己的堆空间中，操作系统在用户进程的堆空间找到可用的内存后，返回该内存空间的起始地址。我们也要实现堆内存管理，为了实现简单，现在并没有为堆单独规划起始地址，而是由用户进程自己的虚拟内存池统一管理，用户进程被加载到内存后，剩余未用的高地址都被作为堆和栈的共享空间。下面我们介绍用户进程虚拟内存池的创建。

为用户进程创建虚拟内存池的函数是 create_user_vaddr_bitmap，它接受一个参数 user_prog，表示用户进程，函数功能是创建用户进程的虚拟地址位图 user_prog->userprog_vaddr，也就是按照用户进程的虚拟内存信息初始化位图结构体 struct virtual_addr。

user_prog->userprog_vaddr.vaddr_start 是位图中所管理的内存空间的起始地址，我们为用户进程定的起始地址是 USER_VADDR_START，该值定义在 process.h 中，其值为 0x8048000，这是 Linux 用户程序入口地址，您可以用 readelf 命令查看一下，大部分可执行程序的“Entry point address”都是在 0x8048000 附近。

变量 bitmap_pg_cnt 用来记录位图需要的内存页框数，计算过程中用到了宏 DIV_ROUND_UP，它用来实现除法的向上取整，此宏定义在 global.h 中，原型是：

```
#define DIV_ROUND_UP(X, STEP) ((X + STEP - 1) / (STEP))
```

接下来通过 get_kernel_pages(bitmap_pg_cnt)为位图分配内存，返回的地址记录在位图指针 user_prog->userprog_vaddr.vaddr_bitmap.bits 中。然后将位图长度记录在 user_prog->userprog_vaddr. vaddr_bitmap.btmp_bytes_len 中。

最后调用函数 bitmap_init(&user_prog->userprog_vaddr.vaddr_bitmap)进行位图初始化，至此用户虚拟地址位图创建完成。

本节中最后要介绍的函数是 process_execute，它接受两个参数，filename 是用户进程地址，name 是进程名，它的功能是创建用户进程 filename 并将其加入到就绪队列等待执行。此函数的实现是类似线程创建的过程，大家应该很熟悉，而且经过前面冗长相关的介绍，我相信此函数不用多说了。

11.3.9 让进程跑起来——用户进程的调度

在这之前，我们已经将进程创建好，并且添加到就绪队列中了，不管任务是线程，还是进程，目前的任务调度器 schedule 一律按内核线程来处理。内核线程是 0 特权级，并且它使用内核的页表，这与进程的区别很大，进程的特权级是 3，并且有自己单独的页表，因此我们需要改进调度器，增加对进程的处理。见代码 11-7。

代码 11-7 （project/c11/b/ thread/thread.c）

```
     …略
   9 #include "process.h"
     …略
 102 /* 实现任务调度 */
 103 void schedule() {
      …略
 121    thread_tag = list_pop(&thread_ready_list);
 122    struct task_struct* next = \
       elem2entry(struct task_struct, general_tag, thread_tag);
```

```
123     next->status = TASK_RUNNING;
124
125     /* 激活任务页表等 */
126     process_activate(next);
127
128     switch_to(cur, next);
129 }
```

这次我们在 thread.c 中需要修改 schedule，修改的内容也很简单，就是在第 126 行增加了代码“process_activate(next);”，process_activate 除了用来更新任务的页表外，还要根据任务是否为进程，修改 tss 中的 esp0，此函数之前已经介绍了。

好啦，本节的内容就这么少，下一节咱们要实际创建用户进程了。

11.3.10　测试用户进程

用户进程的创建是由函数 process_execute 完成的，本节咱们将在 main.c 中调用它，创建两个用户进程。上菜了，请见代码 11-8。

代码 11-8　（project/c11/b/kernel/main.c）

```
   …略
 6 #include "process.h"
 7
 8 void k_thread_a(void*);
 9 void k_thread_b(void*);
10 void u_prog_a(void);
11 void u_prog_b(void);
12 int test_var_a = 0, test_var_b = 0;
13
14 int main(void) {
15     put_str("I am kernel\n");
16     init_all();
17
18     thread_start("k_thread_a", 31, k_thread_a, "argA ");
19     thread_start("k_thread_b", 31, k_thread_b, "argB ");
20     process_execute(u_prog_a, "user_prog_a");
21     process_execute(u_prog_b, "user_prog_b");
22
23     intr_enable();
24     while(1);
25     return 0;
26 }
27
28 /* 在线程中运行的函数 */
29 void k_thread_a(void* arg) {
30     char* para = arg;
31     while(1) {
32        console_put_str("v_a:0x");
33        console_put_int(test_var_a);
34     }
35 }
36
37 /* 在线程中运行的函数 */
38 void k_thread_b(void* arg) {
39     char* para = arg;
40     while(1) {
41        console_put_str(" v_b:0x");
42        console_put_int(test_var_b);
43     }
44 }
45
46 /* 测试用户进程 */
47 void u_prog_a(void) {
48     while(1) {
49        test_var_a++;
50     }
51 }
52
53 /* 测试用户进程 */
```

```
54 void u_prog_b(void) {
55     while(1) {
56        test_var_b++;
57     }
58 }
```

代码 11-8 中，我们在第 18～21 行分别创建了两个线程和两个进程，至此我们的系统中并行了 4 个任务。

先说一下进程吧，第 20 行用 process_execute(u_prog_a, "user_prog_a")创建了用户进程，其中参数 u_prog_a 是用户进程地址，是待运行的进程。它定义在第 47 行，功能是执行死循环，使全局变量 test_var_a++。也许让您意外的是 u_prog_a 是在 main.c 中定义的，而它只是个函数，并不是当初想像的在磁盘上的二进制程序文件。其实……这么做也是没办法啊，我们现在还没有实现文件系统，无法按常规的方法先把程序从磁盘上载入，再解析 elf 文件，再分配内存，再去执行……还是等以后完成了文件系统咱们再那样做吧。不能一口吃个胖子，现在只要能模拟用户进程就足矣了，至少此处用函数来代替用户进程真的够用了。

也许您又在想，你能不能模拟得“像样一些”，好歹专门建立个文件来写这两个用户函数吧，比如叫 user_prog_test.c，那样的话我更容易接受一些，这样把 u_prog_a 挤在 main.c 中，我怎么觉得 u_prog_a 属于内核呢？

先别着急，您看，无论用户进程是从磁盘上载入，还是定义在另外一个文件中，用户进程要执行的话，最终都得在内存中才行。在内存中的用户进程只是一段指令的起始地址而已，我们这里用函数来代替程序文件，无非是少了程序加载的过程，在运行时，函数和从磁盘上载入的程序文件在本质上没区别。如果将它定义在其他文件中，不是还得再单独编译、链接吗？有点麻烦。另外，您的疑虑是正确的，u_prog_a 的地址是在 0xc0000000 以上，位于内核空间，但这并不表示它无法模拟用户进程。其地址如图 11-22 所示。

```
[work@localhost b]$ nm build/kernel.bin |grep -P 'u_prog|test_var'
c00065e0 B test_var_a
c0006480 D test_var_b
c00015df T u_prog_a
c00015f1 T u_prog_b
[work@localhost b]$
```

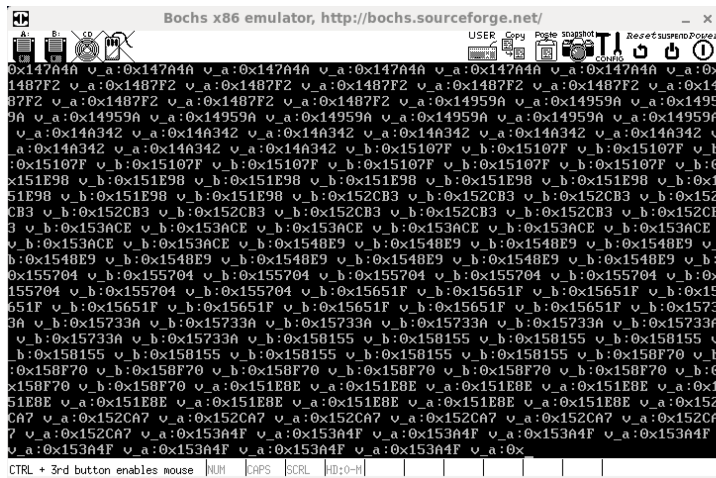
▲图 11-22　“用户进程”地址在内核空间

u_prog_a 的地址是 0xc00015df，按地址来说，它确实是内核空间的函数，但这并不是说此函数就不能模拟用户进程了，只要处理器在执行用户进程时能够访问该地址就行，虽然它属于内核的地址，但我们之前已经在 GDT 中准备了 DPL 为 3 特权级的代码段描述符和数据段描述符，用户进程的 CPL 为 3，并且用指向 DPL 为 3 的段描述符的选择子去访问内核空间，这符合特权级检查。这就像进火车站，军人可以走军人专用通道，普通老百姓就要走一般通道，两种方式都可以上火车，只要能够上火车，火车上的资源谁都可以使用，不会因为是军人就比一般人优先。当然这只是比喻，咱们再从理论方面分析下，想想看，用户进程的特征是什么？3 特权级，自己单独的页表，有 3 特权级的栈，我们的函数 u_prog_a 在运行后完全具备这三个条件。再者，处理器并不是靠代码地址来区分是内核，还是用户进程的，在地址空间上区分内核和用户进程，这是咱们为了方便管理而人为规划的，并不是硬件的规定，只要我们高兴的话，甚至可以把 4GB 空间的高 3GB 内存置为内核空间，只留最低 1GB 内存给用户进程，因此，无论代码位于内核空间，还是用户空间，它都只是一段指令，执行过程中处理器无法区分它是用户代码，还是内核代码，而且也没必要区分，用户态和内核态这两种名词是人们为了方便开发、学习操作系统而提出的概念，目的是为了让大家对处理器执行流所处的某种状态达成一致的理解，因此您可以认为处理器不知道什么是用户态、内核态，它只知道它原生支持的东西，如特权级。若把咱们所说的用户态和内核态的概念转换为处理器能够识别的东西，那就是特权级变化和特权级检查。处理器关注的是只要处理器的当前特权级 CPL 为 3，就不能访问比它特权级更高（数值上 DPL <CPL）的内存。尽管 u_prog_a 本身是在内核地址空间，处理器用的是进程页目录表中指向内核空间的第 768 个页目录项对应的页表来访问的 u_prog_a 及变量 test_var_a，而且页表中低 3GB 的空间没用上，但这和用户态、内核态无关。当用户进程 u_prog_a 运行后，它的特权级为 3，它拥有自己独立的页表，并且是通过为用户进程准备的 DPL 为 3 的段描述符访问内存，这完全符合用户进程的特征和行为，而且最最重要的是我们早已把所有页目录项和页表项的 US 位都置为 1（loader.S 和 memory.c 中所有涉及到 PDE 和 PTE 的地方都用的是 PG_US_U），这表示处理器允许所有特权级的任务可以访问目录项或页表项指向的内存，所以用内核空间中的函数来模拟用户进程是没有问题的。提醒一下大伙儿，如果此时将 US 置为 0 的话咱们就不能用内核函数来模拟用户

进程了，处理器会抛出 page_fault 异常，是的，就是缺页异常，并不是一般保护性异常（GP 异常）。

线程的创建大伙儿肯定再熟悉不过了，那为什么我还要创建线程 k_thread_a 和 k_thread_b 呢？您看，我们创建的两个用户进程 u_prog_a 和 u_prog_b，它们分别将全局变量 test_var_a 和 test_var_b 自增，为了让大家看到用户进程确实有在执行，我想把变量值打印出来，如果变量值一直在变化就说明我们的用户进程一直在执行。但是由于目前我们的用户进程无法直接访问 0 特权级的显存段，故调用打印函数时处理器会抛出一般保护性异常，因此，要么将显存段的 DPL 和 eflags 寄存器的 IOPL 改为 3，要么在 bochs 中通过调试来查看，或者干脆由高特权级的线程来帮忙打印。这里用的就是最后一个方法，为了让大伙儿看到用户进程确实在运行，用 k_thread_a 和 k_thread_b 分别将变量 test_var_a 和 test_var_b 的值打印出来。

好啦，在一番“晓之以理，动之以情”之后，我们要查看下运行结果啦，编译、写入磁盘后，运行结果如图 11-23 所示。



▲图 11-23　用户进程的执行

运行一段时间之后，这两个变量的值已经很大了，本节的用户程序暂时成功了。

为了确认一下用户进程确实进入了 3 特权级，咱们再调试一下，u_prog_a 的地址是 0xc00015df，我们可以在此设置断点，然后查看 cs 寄存器的值，如图 11-24 所示。

```
<bochs:1> lb 0xc00015df
<bochs:2> c
(0) Breakpoint 1, 0xc00015df in ?? ()
Next at t=21937798
(0) [0x0000000015df] 002b:c00015df (unk. ctxt): push ebp              ; 55
<bochs:3> sreg
es:0x0033, dh=0x00cff300, dl=0x0000ffff, valid=1
        Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed
cs:0x002b, dh=0x00cff900, dl=0x0000ffff, valid=1
        Code segment, base=0x00000000, limit=0xffffffff, Execute-Only, Non-Confo
rming, Accessed, 32-bit
ss:0x0033, dh=0x00cff300, dl=0x0000ffff, valid=1
        Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed
ds:0x0033, dh=0x00cff300, dl=0x0000ffff, valid=1
        Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed
fs:0x0033, dh=0x00cff300, dl=0x0000ffff, valid=1
        Data segment, base=0x00000000, limit=0xffffffff, Read/Write, Accessed
gs:0x0000, dh=0x00001000, dl=0x00000000, valid=0
ldtr:0x0000, dh=0x00008200, dl=0x0000ffff, valid=1
tr:0x0020, dh=0xc0808b00, dl=0x67c0006b, valid=1
gdtr:base=0xc0000900, limit=0x37
idtr:base=0xc0006600, limit=0x17f
<bochs:4>
```

▲图 11-24　用户进程在 3 特权级调试

您看，我们用 lb 命令使 bochs 在虚拟地址 0xc00015df 处停下，用命令 c 持续执行，当 bochs 停住后，执行命令 sreg 查看段寄存器，此时 cs 的值为 0x002b，我们关注最低 4 位，其值为 b，换为二进制是 1011，最低 2 位为 rpl，也就是 3，所以可以判断此时用户进程确实是在 3 特权级下，与预期符合。

好，有关用户进程的部分暂且告一段落，以后我们还要在里面加新的内容。

第 12 章　进一步完善内核

12.1 Linux 系统调用浅析

系统调用是这么来的，为方便人们开发程序，一些前辈们开发出一个基础平台，只要人们在这个平台上做少量的工作就能完成各种各样功能的程序，这个基础平台就是操作系统。虽然基于操作系统开发程序使开发效率大大提升了，但人们也因此受到了束缚，必须要遵循操作系统制定的规则。为什么要制定规则呢？一部分原因是为安全起见，不能由程序“乱来”。操作系统不允许用户程序过于“神通广大”，它把用户程序的能力限制得很低，即使用户程序做了坏事，也不会影响到整个计算机的安全。不过，在很多时候用户进程还是需要提升“战斗力”以完成一些合法的工作，比如访问打印机为用户打印文件等。操作系统对用户进程绝对是不放心的，绝对不能把权力放出去，为满足这一合理需求，操作系统说了：“有事您找我，我亲自帮您办。”操作系统提供了一组“提升战斗力”的接口，每个接口都是不同的功能，用户进程需要什么功能就调用什么，这组接口就是系统调用接口。

系统调用就是让用户进程申请操作系统的帮助，让操作系统帮其完成某项工作，也就是相当于用户进程调用了操作系统的功能，因此“系统调用”准确地来说应该被称为“操作系统功能调用”。

如今 Linux 的发展如火如荼，融聚了众多精英的智慧，几乎任何大型互联网公司的底层业务都要用 Linux 来支撑，因此咱们还是参照 Linux 系统调用的原理，模仿着它咱们实现一份简易的系统调用版本（话说在咱们系统中什么都是简易的）。

Linux 系统调用是用中断门来实现的，通过软中断指令 int 来主动发起中断信号。由于要支持的系统功能很多，总不能一个系统功能调用就占用一个中断向量，真要是这样的话整个中断描述符表都不够用呢。Linux 只占用一个中断向量号，即 0x80，处理器执行指令 int 0x80 时便触发了系统调用。为了让用户程序可以通过这一个中断门调用多种系统功能，在系统调用之前，Linux 在寄存器 eax 中写入子功能号，例如系统调用 open 和 close 都是不同的子功能号，当用户程序通过 int 0x80 进行系统调用时，对应的中断处理例程会根据 eax 的值来判断用户进程申请哪种系统调用。

我们先来看看 Linux 中系统调用 syscall 是怎么用的。在 Linux 中最方便的帮助就是 man 了，还是老样子，先执行 man syscall 回车，看看有什么惊喜，输出结果如图 12-1 所示。

```
NAME
       syscall - indirect system call

SYNOPSIS
       #define _GNU_SOURCE        /* or _BSD_SOURCE or _SVID_SOURCE */
       #include <unistd.h>
       #include <sys/syscall.h>   /* For SYS_xxx definitions */

       int syscall(int number, ...);

DESCRIPTION
       syscall() performs the system call whose assembly language
 interface has the specified number with the specified arguments.
       Symbolic constants for system calls can be found in the header
file <sys/syscall.h>.

RETURN VALUE
       The return value is defined by the system call being invoked.
In general, a 0 return value indicates success.  A -1 return
       value indicates an error, and an error code is stored in errno.

NOTES
       syscall() first appeared in 4BSD.

EXAMPLE
       #define _GNU_SOURCE
       #include <unistd.h>
       #include <sys/syscall.h>
       #include <sys/types.h>

       int
       main(int argc, char *argv[])
       {
           pid_t tid;

           tid = syscall(SYS_gettid);
       }
```

▲图 12-1　syscall 的 man 帮助

syscall 的原型是 int syscall(int number, …)，其中的 number 是 int 型，这是系统调用号，也就是前面所说的子功能号。不同的子功能需要的参数也是不同的，所以 number 后面的“…”表示此函数支持变参，函数 syscall 支持不同参数个数的系统调用，在新版本 Linux 中，所有的系统调用功能都可通过这一个函数完成。顺便说一

句，函数 syscall 并不是由操作系统提供的，它是由 C 运行库 glibc（GNU 发布的 libc 库版本）提供的，因此 syscall 实际上是库函数，提到这个是为了与后面的宏方式做对比。

在图最下面的框框中，用 syscall(SYS_getpid)举例了 syscall 的用法。此时只给 syscall 代入了一个参数 SYS_getpid，也就是子功能号。SYS_getpid 是个宏，其定义在文件“/usr/include/bits/syscall.h”中，如图 12-2 所示。

SYS_getpid 的值是__NR_getpid，__NR_getpid 还是个宏，在 32 位机器上，定义在文件“/usr/include/asm/unistd_32.h”中，值为 20，输出结果如图 12-3 所示。

```
 95 #define SYS_getpgid __NR_getpgid
 96 #define SYS_getpgrp __NR_getpgrp
 97 #define SYS_getpid __NR_getpid
 98 #define SYS_getpmsg __NR_getpmsg
 99 #define SYS_getppid __NR_getppid
100 #define SYS_getpriority __NR_getpriority
101 #define SYS_getresgid __NR_getresgid
102 #define SYS_getresgid32 __NR_getresgid32
103 #define SYS_getresuid __NR_getresuid
104 #define SYS_getresuid32 __NR_getresuid32
105 #define SYS_getrlimit __NR_getrlimit
106 #define SYS_getrusage __NR_getrusage
107 #define SYS_getsid __NR_getsid
108 #define SYS_gettid __NR_gettid
109 #define SYS_gettimeofday __NR_gettimeofday
110 #define SYS_getuid __NR_getuid
111 #define SYS_getuid32 __NR_getuid32
112 #define SYS_getxattr __NR_getxattr
/usr/include/bits/syscall.h [RO]
```

▲图 12-2　syscall.h

```
28 #define __NR_getpid          20
29 #define __NR_mount           21
30 #define __NR_umount          22
31 #define __NR_setuid          23
32 #define __NR_getuid          24
33 #define __NR_stime           25
34 #define __NR_ptrace          26
35 #define __NR_alarm           27
36 #define __NR_oldfstat        28
37 #define __NR_pause           29
38 #define __NR_utime           30
/usr/include/asm/unistd_32.h [RO]
```

▲图 12-3　unistd_32.h

此处“__NR_”+“系统调用名称”形式的子功能号宏还会被用到，一会儿我们会再提起，总之对用户进程而言，在 Linux 上执行系统调用，只需要提供子功能号和参数就行了，我们也不能给用户进程添麻烦，也只要求提供这两项。

我们再回头看一下图 12-1，在其上面用方框框起来的是“indirect system call”，这是说 syscall 属于“间接”的系统调用，其实想想这种“间接”也是必然的，因为它是由 glibc 提供的库函数，您懂的，使用库函数的好处有很多，其中之一就是实现跨平台兼容，因此库函数几乎都是对宿主系统的系统调用接口的封装。说到这您一定想到了，肯定还有“直接”进行系统调用的方式，这个直接的方式肯定不再是库函数了，应该是由操作系统提供，这样才显得“直接”。您说对了，这个直接的做法是利用操作系统提供的_syscall[X]，它是一系列的宏，这次我们执行 man _syscall 回车，输出结果如图 12-4 所示。

```
NAME
       _syscall - invoking a system call without library support (OBSOLETE)

SYNOPSIS
       #include <linux/unistd.h>

       A _syscall macro

       desired system call

DESCRIPTION
       The  important  thing to know about a system call is its prototype.  You need to
know how many arguments, their
       types, and the function return type.  There are seven macros that make the actual
 call into the system  easier.
       They have the form:

              _syscallX(type,name,type1,arg1,type2,arg2,...)

       where

              X is 0-6, which are the number of arguments taken by the system call

              type is the return type of the system call

              name is the name of the system call

              typeN is the Nth argument's type

              argN is the name of the Nth argument
```

▲图 12-4　_syscall 的 man 帮助

有关_syscall 的几个关键点我已在图中用框框标出，咱们从上到下说明。最上面的英文表示，用_syscall 进行系统调用不需要通过库函数，这说明它是直接的系统调用方式。不过，在它的后面的括号中有个单词 OBSOLETE，意思是过时的、废弃的，这说明此方法已经被 Linux 废弃了（注意，只是废弃了_syscall 这个符号，后面在源码解析后会再阐述），废弃的原因是此方式最多支持 6 个参数，一会儿您就了解了。_syscall 虽然已经过时了，但它的实现思路非常简单，对于咱们简易版本的系统调用需求已经绰绰有余，所以还是借鉴这位老前辈的方法为己用吧，磨刀不误砍柴工，下面我们花点时间学习一下它的实现。

_syscall 是系统调用“族”，所以图中用_syscallX 来表示它们，其中的 X 表示系统调用中的参数个数，其原型是_syscallX(type,name,type1,arg1,type2,arg2,…)。_syscallX 是用宏来实现的，根据系统调用中参数个数、类型及返回值的不同，这里共有 7 个不同的宏，分别是_syscall[0-6]，因此，对于参数个数不同的系统调用，需要调用不同的宏来完成。图中对参数已有介绍，type 是系统调用的返回值类型，name 是系统调用名称（字符串），最后通过宏替换会转换成数值型的子功能号，typeN 和 argN 配对出现，分别表示参数的类型及变量名。下面咱们拿_syscall3 举例，其他同理，先看一下宏具体实现，如下所示。

```
1 #define _syscall3(type, name, type1, arg1, type2, arg2, type3, arg3) \
2 type name(type1 arg1, type2 arg2, type3 arg3) { \
3  long __res; \
4  __asm__ volatile ("push %%ebx; movl %2,%%ebx; int $0x80; pop %%ebx" \
5           : "=a" (__res) \
6           : "0" (__NR_##name),"ri" ((long)(arg1)),"c" ((long)(arg2)), \
7             "d" ((long)(arg3)) : "memory"); \
8  __syscall_return(type,__res); \
9 }
```

此宏不长，一共 9 行。其中第 1 行是宏名及参数的定义，完全是按照图 12-4 中的说明实现的。

这里给大伙说明下，Linux 中的系统调用是用寄存器来传递参数的，这些参数需要按照从左到右的顺序依次存入到不同的通用寄存器（除 esp）中。其中，寄存器 eax 用来保存子功能号，ebx 保存第 1 个参数，ecx 保存第 2 个参数，edx 保存第 3 个参数，esi 保存第 4 个参数，edi 保存第 5 个参数。传递参数还可以用栈（内存），不知道您想过没有，为什么 Linux 用寄存器来传递参数，而不用栈？用寄存器快？肯定是这样的，没有哪个操作系统愿意更慢。不过这个“快”可不是出于存储介质方面的考虑，而是用寄存器传参的步骤少一些，听我慢慢道来。用户进程执行 int 0x80 时还处于用户态，编译器根据 c 调用约定，系统调用所用的参数会被压到用户栈中，这是 3 特权级栈。当 int 0x80 执行后，任务陷入内核态，此时进入了 0 特权级，因此需要用到 0 特权级栈，但系统调用的参数还在 3 特权级的栈中，为了获取用户栈地址，还得在 0 特权级栈中获取处理器自动压入的用户栈的 SS 和 esp 寄存器的值，然后再次从用户栈中获取参数。您看，光传递参数就涉及到了多次内存访问的情况，内存比寄存器要慢，而且步骤很麻烦，我会在最后给您提供个栈传递参数的例子，到时候您可以体会一下。

回到代码，第 2～9 行是宏体，这是按照函数定义的方式定义的。第 2 行的 type 是函数的返回值类型，name 是函数名，也就是系统调用名，函数名后面括号中是一系列的形参，用的是宏_syscall3 中的参数。第 3 行是函数体的开始，__res 是返回值。

跨过第 4 行先说下第 5～7 行，第 5 行的"=a" (__res)位于输出部 output，这表明变量__res 由寄存器 eax 赋值。我们知道，根据 abi 约定，eax 作为函数调用的返回值，这里是用变量__res 来存储从中断返回后的返回值。第 6 行是参数输入部 input，"0" (__NR_##name)中的_NR_##name 是系统调用的字符串名，经过预处理后会先变成__NR_系统调用名，然后变成数值型的子功能号，其中“##”表示联结字符串，这是预处理器支持的语法，咱们在介绍中断入口函数时已经说过了，比如系统调用名为 getpid，预处理后就变成__NR_getpid，__NR_getpid 也是宏，它的值如前面的图 12-3 所示。"0" (__NR_##name)中的 0 是通用约束，表示__NR_##name 使用的寄存器或内存与第 0 个约束表达式使用的寄存器或内存一致，这里指的是和第 5 行的"=a" (__res)一致，也就是寄存器 eax 既做子功能号输入，又做返回值的输出。后面"ri" ((long)(arg1))是将变量 arg1 约束到通用寄存器中，"c" ((long)(arg2))是将变量约束到 ecx 寄存器中，第 7 行的"d" ((long)(arg3))是将变量约束到 edx 中。

再回头看第 4 行，此行是内联汇编代码，其中 push %%ebx 的作用是在用户空间的栈中提前保护好 ebx 的值，movl %2,%%ebx 将 arg1 的值写入寄存器 ebx，%2 是序号占位符，表示第 2 个约束，即 arg1 对应的寄存器或内存。int $0x80 触发软中断，进行系统调用，完成后通过 pop %%ebx 恢复 ebx 的值。

第 8 行是__syscall_return(type,__res);，对返回值__res 判断后返回，其中__syscall_return 也是个宏，实现如下，不再说明。

```
1 #define __syscall_return(type, res)                    \
2    do {                           \
3     if ((unsigned long)(res) >= (unsigned long)(-125)) { \
4           errno = -(res);                          \
5           res = -1;                             \
6    }                                        \
7    return (type) (res);                           \
8 } while (0)
```

当参数多于 5 个时，可以用内存来传递，注意啦，此时在内存中存储的参数仅是第 1 个参数及第 6 个以上的所有参数，不包括第 2～5 个参数，第 2～5 个参数依然要顺序放在寄存器 ecx、edx、esi 及 edi 中，eax 始终是子功能号。我们看下宏_syscall6 的实现就清楚了，如下所示。

```
 1 #define _syscall6(type,name, type1,arg1, type2,arg2, type3,arg3,
 2                             type4,arg4, type5,arg5, type6,arg6) \
 3 type name (type1 arg1,type2 arg2,type3 arg3,\
 4             type4 arg4,type5 arg5,type6 arg6) { \
 5    long __res; \
 6    struct { long __a1; long __a6; } __s = { (long)arg1, (long)arg6 }; \
 7    __asm__ volatile ("push %%ebp ; push %%ebx ; movl 4(%2),%%ebp ; " \
 8                      "movl 0(%2),%%ebx ; movl %1,%%eax ; int $0x80 ; " \
 9                      "pop %%ebx ;  pop %%ebp" \
10    : "=a" (__res) \
11    : "i" (__NR_##name),"0" ((long)(&__s)),"c" ((long)(arg2)), \
12    "d" ((long)(arg3)),"S" ((long)(arg4)),"D" ((long)(arg5)) \
13    : "memory"); \
14    __syscall_return(type,__res); \
15 }
```

大体上和_syscall3 的原理差不多，无非是为了存储第 6 个参数，在_syscall6 的第 6 行声明了结构及实例 struct { long __a1; long __a6; } __s，然后在第 11 行将实例__s 的地址作为输入，最后在第 7 行用 movl 4(%2),%%ebp 将第 6 个参数写入寄存器 ebp，在第 8 行用 movl 0(%2),%%ebx 将第 1 个参数写入 ebx。总之 ebx 中还是第 1 个参数。

好啦，介绍就到这，总结一下。

宏_syscall 和库函数 syscall 相比，syscall 实现更灵活，对用户来说任何参数个数的系统调用都统一用一种形式，用户只要记住 syscall 就可以了，而宏_syscall 的实现比较死板，针对每种参数个数的系统调用都要有单独的形式，因此支持的参数数量必然有限，而且用户要记住 7 种形式_syscall[0-6]，调用时除了输入实参外，还要输入实参的类型，确实有些麻烦，此外这个宏会引发安全漏洞（有兴趣可自行检索相关资料），故必然会被 syscall 取代。

强调一下，这里所说 Linux 废弃宏_syscall，是指 Linux 系统不提供_syscall 的定义及实现了，因此用户进程无法通过宏_syscall 进行系统调用（当然咱们可以再写一个），_syscall 虽然已经很直接了，但真正最直接的方式是汇编代码“mov eax，子功能号;int 0x80”，_syscall 是这两句汇编指令的封装，在 Linux 系统内部，系统调用接口并未改变，依然是在中断向量 0x80 对应的中断处理例程中把 eax 的值作为子功能号，然后调用相应的子功能函数，其对外形式依旧是 eax 为子功能号，然后执行 0x80 中断。说白了就是 Linux 中虽然没有方便的_syscall 了，但我们依然可以写汇编代码“mov eax, 子功能号;int 0x80”去执行 Linux 系统调用。

虽然库函数 syscall 更加先进，但咱们为了实现简单，还是参考这个“过气”的宏_syscall 来完成咱们的工作吧，毕竟我能力有限，短期内不能凭一己之力拿下全世界的“智慧结晶”，有关这个系统调用的原理就到此结束啦。

12.2 系统调用的实现

上节已经跟大伙儿介绍了 Linux 系统调用的部分实现，本节以它为范本，参照着实现自己的系统调用。

12.2.1　系统调用实现框架

一个系统功能调用分为两部分，一部分是暴露给用户进程的接口函数，它属于用户空间，此部分只是用户进程使用系统调用的途径，只负责发需求。另一部分是与之对应的内核具体实现，它属于内核空间，此部分完成的是功能需求，就是我们一直所说的系统调用子功能处理函数。为区分这两部分，一般情况下内核空间的函数名要在用户空间函数名前加“sys_”。咱们以函数 getpid 为例说明，我们知道 getpid 的功能是返回任务的 pid，这是给用户进程使用的系统调用的接口，接口的另一端是实现该功能的内核函数，该内核函数就是系统调用子功能为“getpid”对应的函数，即 sys_getpid，它才是幕后功臣，由它负责找出调用者的 pid 并返回。

前面已经介绍过部分 Linux 中系统调用相关的内容，大家也大致了解系统调用实现的框架，现在到了咱们动手实践的时候了，先梳理下咱们系统调用的实现思路。

（1）用中断门实现系统调用，效仿 Linux 用 0x80 号中断作为系统调用的入口。

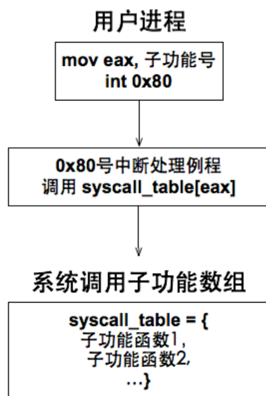
（2）在 IDT 中安装 0x80 号中断对应的描述符，在该描述符中注册系统调用对应的中断处理例程。

（3）建立系统调用子功能表 syscall_table，利用 eax 寄存器中的子功能号在该表中索引相应的处理函数。

（4）用宏实现用户空间系统调用接口_syscall，最大支持 3 个参数的系统调用，故只需要完成_syscall[0-3]。寄存器传递参数，eax 为子功能号，ebx 保存第 1 个参数，ecx 保存第 2 个参数，edx 保存第 3 个参数。

以上 4 个步骤的流程如图 12-5 所示。

看上去还是蛮简单的，大方向就是这样，出发。



▲图 12-5　系统调用实现流程

12.2.2　增加 0x80 号中断描述符

为实现系统调用，我们需要改进相关的一系列文件，我们有关中断处理例程的文件有 kernel.S 和 interrupt.c，一步一步来吧，首先我们要修改 interrupt.c，在其中安装 0x80 对应的中断描述符，请见代码 12-1。

代码 12-1　（project/c12/a/kernel/interrupt.c）

```
   …略
12 #define IDT_DESC_CNT 0x81     // 目前总共支持的中断数
   …略
17 extern uint32_t syscall_handler(void);
   …略
72 /*初始化中断描述符表*/
73 static void idt_desc_init(void) {
74     int i, lastindex = IDT_DESC_CNT - 1;
75     for (i = 0; i < IDT_DESC_CNT; i++) {
76          make_idt_desc(&idt[i], IDT_DESC_ATTR_DPL0, intr_entry_table[i]);
77     }
78 /* 单独处理系统调用，系统调用对应的中断门 dpl 为 3，
79  * 中断处理程序为单独的 syscall_handler */
80     make_idt_desc(&idt[lastindex], IDT_DESC_ATTR_DPL3, syscall_handler);
81     put_str("   idt_desc_init done\n");
82 }
```

代码第 12 行将宏 IDT_DESC_CNT 修改为 0x81，这表示我们最大支持 0x81 个中断，即 0～0x80，0x80 是我们系统调用对应的中断向量。

第 17 行声明了外部函数 syscall_handler，我们将在 kernel.S 中定义它，syscall_handler 就是系统调用

对应的中断入口例程。

在后面的 idt_desc_init 函数中，我们在第 80 行增加了 0x80 号中断向量对应的中断描述符，在描述符中注册的中断处理例程为 syscall_handler。这里要注意的是记得给此描述符的 dpl 指定为用户级 IDT_DESC_ATTR_DPL3，若指定为 0 级，则在 3 级环境下执行 int 指令会产生 GP 异常。

好啦，interrupt.c 就修改完了。

12.2.3　实现系统调用接口

之前我们已经讨论了宏_syscall 的原理，用户进程可以通过调用宏_syscall[0-6]进行系统调用，它的核心就是用内联汇编传参并触发中断。我们说过了，选择它的原因就是因为它简单，现在咱们就要参照它实现自己的版本啦，告诉大伙儿一个好消息，咱们是一山更比一山低，咱们的实现比它还要简单。

这次我们把它定义在 syscall.c 中，它在目录 lib/user/下，请大伙儿见代码 12-2。

代码 12-2　（project/c12/a/lib/user/syscall.c）

```
 1 #include "syscall.h"
 2
 3 /* 无参数的系统调用 */
 4 #define _syscall0(NUMBER) ({            \
 5    int retval;                          \
 6    asm volatile (                       \
 7    "int $0x80"                       \
 8    : "=a" (retval)                      \
 9    : "a" (NUMBER)                       \
10    : "memory"                          \
11    );                                  \
12    retval;                            \
13 })
14
   …略
39 /* 三个参数的系统调用 */
40 #define _syscall3(NUMBER, ARG1, ARG2, ARG3) ({       \
41    int retval;                                      \
42    asm volatile (                                   \
43       "int $0x80"                                   \
44       : "=a" (retval)  \
45       : "a" (NUMBER), "b" (ARG1), "c" (ARG2), "d" (ARG3) \
46       : "memory"        \
47    );                                       \
48    retval;                                  \
49 })
```

如前所述，咱们打算最多支持 3 个参数的系统调用，它们是_syscall[0-3]，代码 12-2 中列出了无参数版本和 3 个参数的版本，由于_syscall[1-3]只是在为参数赋值时有所不同，故只贴出了_syscall3。

咱们的_syscall 版本和 Linux 的版本类似，不过和它相比，咱们的版本更简单一些，Linux 中是用宏定义了一个函数，咱们这里是直接用大括号完成的，也许有同学对大括号的这种用法比较陌生，大括号中最后一个语句的值会作为大括号代码块的返回值，而且要在最后一个语句后添加分号';'，否则编译时会报错。另外，在咱们的内联汇编中都没用到通用约束，确实简陋了很多。

想必在介绍系统调用原理的时候大伙已经对_syscall 非常清楚了，因此本节又没戏唱了，大伙儿下节再见。

12.2.4　增加 0x80 号中断处理例程

下面需要修改 kernel.S，在里面安装中断向量 0x80 对应的中断处理程序，也就是上节提到的 syscall_handler，请见代码 12-3。

代码 12-3　（project/c12/a/kernel/kernel.S）

```
 98 ;;;;;;;;;;;;;;;;   0x80 号中断   ;;;;;;;;;;;;;;;;
 99 [bits 32]
100 extern syscall_table
101 section .text
102 global syscall_handler
```

```
103 syscall_handler:
104 ;1 保存上下文环境
105    push 0         ; 压入0，使栈中格式统一
106
107    push ds
108    push es
109    push fs
110    push gs
111    pushad         ; PUSHAD指令压入32位寄存器，其入栈顺序是:
112            ; EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI
113
114    push 0x80      ; 此位置压入0x80也是为了保持统一的栈格式
115
116 ;2 为系统调用子功能传入参数
117    push edx       ; 系统调用中第3个参数
118    push ecx       ; 系统调用中第2个参数
119    push ebx       ; 系统调用中第1个参数
120
121 ;3 调用子功能处理函数
122    call [syscall_table + eax*4]
123    add esp, 12             ; 跨过上面的三个参数
124
125 ;4 将call调用后的返回值存入当前内核栈中eax的位置
126    mov [esp + 8*4], eax
127    jmp intr_exit           ; intr_exit返回，恢复上下文
```

在第100行中声明了外部数据结构syscall_table，syscall_table是个数组，数组成员是系统调用中子功能对应的处理函数（以后将在新文件中定义它），这里我们用子功能号在此数组中索引子功能号对应的处理函数。

第103行是中断例程syscall_handler的定义，为了复用intr_exit，此例程的前半部分和其他中断例程一样，第105行压入了中断错误码0，第107～111行用来保存任务的上下文，第114行显式压入了中断号0x80。其实第105行和第114行只是占位用，无所谓内容。不过还是为了意义统一，分别压入了错误码和中断向量号。

第117～119行是为子功能函数准备参数，由于只支持3个参数的系统调用，故只压入了三个参数，按照C调用约定，最右边的参数先入栈，因此先把edx中的第3个参数入栈，其次是ecx中的第2个参数、ebx中的第1个参数。注意，这里我们不管具体系统调用中的参数是几个，一律压入3个参数，也许您对此感到奇怪。是这样的，子功能处理函数都有自己的原型声明，声明中包括参数个数及类型，编译时编译器会根据函数声明在栈中匹配出正确数量的参数，进入函数体后，根据C调用约定，栈顶的4字节（32位系统，下同）是函数的返回地址，往上（高地址的栈底方向）的4字节是第1个参数，再往上的4字节便是第2个参数，依此类推。在函数体中，编译器生成的取参数指令是从栈顶往上（跨过栈顶的返回地址，向高地址方向）获取参数的，参数个数是通过函数声明事先确定好的，因此并不会获取到错误的参数，从而保证了多余的参数用不上，因此，尽管我们压入了3个参数，但对于那些参数少于3个的函数也不会出错，而我们也只是浪费了一点点栈空间。

寄存器eax中是系统调用子功能号，用它在数组syscall_table中索引对应的子功能处理函数。syscall_table中存储的是函数地址，每个成员是4字节大小，因此在第122行中，要用eax*4做syscall_table的偏移量，这样代码“call [syscall_table + eax*4]”便去调用子功能处理函数。调用之后，在第123行通过“add esp, 12”跨过这三个参数。

根据二进制编程接口abi约定，寄存器eax用来存储返回值。经过上面第122行的call函数调用，如果有返回值的话，eax的值已经变成了返回值（如果没有返回值也没关系，编译器会保证函数返回后eax的值不变），此时我们要把返回值传给用户进程，但是从内核态退出时，要从内核栈中恢复寄存器上下文，这会将当前eax的返回值覆盖，那如何将返回值传给用户进程呢？聪明的您一定想到了，就是把寄存器eax的值回写到内核栈中用于保存eax的内存处，这样从内核返回时，popd指令也只是用该返回值重新覆盖一次eax寄存器，返回到用户态时，用户进程便获取到了系统调用函数的返回值。

以上的思路是在第126行通过“mov [esp + 8*4], eax”实现的，此行代码就是将返回值写到了栈（此时是内核栈）中保存eax的那个内存空间。这里解释一下[esp+8*4]，这是寄存器相对寻址，esp就是当前栈顶，8*4就是相对栈顶，往栈中高地址方向的偏移量，其实把8*4拆分成(1+7)*4更好，其中的1是指上面的push 0x80所占的4字节，另外的7是指pushad指令会将eax最先压入，故要跨过7个4字节，总

共是 8 个 4 字节，即[esp+8*4]是对应栈中 eax 的“藏身之所”。

第 127 行通过“jmp intr_exit”从中断出口函数 intr_exit 返回，这里是复用 intr_exit，它还是老版本，没有变化。

好啦，本节到此结束啦，离完成还有不远的距离，大伙儿辛苦了。

12.2.5　初始化系统调用和实现 sys_getpid

为了支持系统调用，我们的前期工作做得差不多了：已经在 IDT 中安装了 0x80 号中断描述符，增加了相应的中断处理例程，实现了_syscall 接口，似乎还差那么一点点，对，我们还需要个数据结构，系统调用子功能数组 syscall_table，用它来存放不同子功能对应的处理函数。

实现 syscall_table 还是很简单的，就是数组定义，两句话就完成了，现在还需要在这个数组中注册子功能处理函数，也就是使数组 syscall_table 中的元素为子功能处理函数指针，这样在 0x80 号中断向量的中断处理例程中才能调用到相应的子功能处理函数，还记得吗，具体的函数调用是文件 kernel.S 第 122 行的代码“call [syscall_table + eax*4]”。在咱们的设计中，往 syscall_table 中注册处理函数这项工作是在初始化系统调用时完成的。可现在的关键是目前没有实现具体系统调用功能，还没有子功能处理函数呢，巧妇难为无米之炊，要不趁现在一块定义个子功能处理函数吧。

咱们要实现的第一个系统调用是 getpid，getpid 的功能是获取任务自己的 pid，getpid 是给用户进程使用的接口函数，它在内核中对应的处理函数是 sys_getpid，具体请见代码 12-4。

代码 12-4　（project/c12/a/userprog/syscall-init.c）

```
   …略
 7 #define syscall_nr 32
 8 typedef void* syscall;
 9 syscall syscall_table[syscall_nr];
10
11 /* 返回当前任务的 pid */
12 uint32_t sys_getpid(void) {
13     return running_thread()->pid;
14 }
15
16 /* 初始化系统调用 */
17 void syscall_init(void) {
18     put_str("syscall_init start\n");
19     syscall_table[SYS_GETPID] = sys_getpid;
20     put_str("syscall_init done\n");
21 }
```

代码 12-4 的第 7～9 行定义了 syscall_table 相关参数，syscall_nr 表示最大支持的系统调用子功能个数，其值为 32。第 8 行用 typedef 自定义 syscall 类型为空指针 void*，第 9 行 syscall 是数组 syscall_table 的元素类型，也就是 syscall_table 为函数指针数组。

第 12 行是 sys_getpid 的定义，它的实现很简单，就是将当前任务 pcb 中的 pid 返回。不过此时咱们的任务创建中还没有分配 pid 的功能，先不急，一会儿再把相关代码补上。

第 17 行是初始化系统调用函数 syscall_init，很简单，就是为数组 syscall_table 赋值，这里用到了 SYS_GETPID，它是个枚举型数值，表示系统调用子功能号，目前其值为 0，定义在 lib/user/syscall.h 中，后面咱们会介绍。

下面补上为任务分配 pid 相关的代码，先见代码 12-5。

代码 12-5　（project/c12/a/thread/thread.h）

```
   …略
10 typedef int16_t pid_t;
   …略
76 /* 进程或线程的 pcb，程序控制块 */
77 struct task_struct {
78     uint32_t* self_kstack;   // 各内核线程都用自己的内核栈
79     pid_t pid;
80     enum task_status status;
       …略
```

代码 12-5 仅在 struct task_struct 中添加了成员 pid，其类型为 int16_t。下面看 thread.c 中分配 pid 的相关代码。

代码 12-6 （project/c12/a/thread/thread.c）

```
   …略
15 struct lock pid_lock;            // 分配 pid 锁
   …略
35 /* 分配 pid */
36 static pid_t allocate_pid(void) {
37     static pid_t next_pid = 0;
38     lock_acquire(&pid_lock);
39     next_pid++;
40     lock_release(&pid_lock);
41     return next_pid;
42 }
   …略
58 /* 初始化线程基本信息 */
59 void init_thread(struct task_struct* pthread, char* name, int prio) {
60     memset(pthread, 0, sizeof(*pthread));
61     pthread->pid = allocate_pid();
62     strcpy(pthread->name, name);
       …略
170 /* 初始化线程环境 */
171 void thread_init(void) {
172     put_str("thread_init start\n");
173     list_init(&thread_ready_list);
174     list_init(&thread_all_list);
175     lock_init(&pid_lock);
176 /* 将当前 main 函数创建为线程 */
177     make_main_thread();
178     put_str("thread_init done\n");
179 }
```

本次 thread.c 中涉及的变化有第 15 行的 struct lock pid_lock，这是定义了 pid 锁，pid 必须是唯一的，此锁用来在分配 pid 时实现互斥，避免为不同的任务分配重复的 pid。第 36 行的函数 allocate_pid 用来分配 pid，这里用静态全局变量 next_pid 的值作为 pid，next_pid 初始为 0，其加 1 后的结果为新线程的 pid，因此第 1 个任务的 pid 为 1（Linux 中 pid 为 0 的任务是 init，将来咱们实现任务 init 后也要把它的 pid 分配为 1），之后任务的 pid 会自增。分配 pid 是在线程创建后的初始化期间进行的，因此函数 allocate_pid 是在 init_thread 函数中使用，具体代码是第 61 行的"pthread->pid = allocate_pid();"。注意啦，pid_lock 的类型是 struct lock，在使用前要初始化，这是在函数 thread_init 中第 175 行的"lock_init(&pid_lock)"完成的。

好啦，相关的修改差不多了，下一步就要为用户进程添加 getpid 系统调用接口，兄弟们休息下，下节继续。

12.2.6 添加系统调用 getpid

本节要在系统中安装第一个系统调用——getpid，需要增加几个文件，直接上菜啦，请见代码 12-7。

代码 12-7 （project/c12/a/lib/user/syscall.h）

```
1 #ifndef __LIB_USER_SYSCALL_H
2 #define __LIB_USER_SYSCALL_H
3 #include "stdint.h"
4 enum SYSCALL_NR {
5    SYS_GETPID
6 };
7 uint32_t getpid(void);
8 #endif
```

在 syscall.h 中主要定义了枚举结构 enum SYSCALL_NR，此结构用来存放系统调用子功能号，目前里面只有 SYS_GETPID，默认值为 0，以后再增加新的系统调用后还需要把新的子功能号添加到此结构中。

接下来要考虑实现系统调用接口了，这里要实现的接口是 getpid，将它定义在哪里呢？想来想去，还是放在 syscall.c 中比较合适，这样比较方便调用宏_syscall[0-3]，如代码 12-8 所示。

代码 12-8　（project/c12/a/lib/user/syscall.c）

```
 1 #include "syscall.h"
 2
 3 /* 无参数的系统调用 */
 4 #define _syscall0(NUMBER) ({          \
 5    int retval;                       \
 6    asm volatile (                    \
 7    "int $0x80"                       \
 8    : "=a" (retval)                  \
 9    : "a" (NUMBER)                    \
10    : "memory"                       \
11    );                            \
12    retval;                          \
13 })
   …略
51 /* 返回当前任务 pid */
52 uint32_t getpid() {
53    return _syscall0(SYS_GETPID);
54 }
```

此次在 syscall.c 的第 51 行添加了 getpid 系统调用，当然您懂的，它只是用户接口，真正实现此功能的函数是位于 syscall-init.c 中的 sys_getpid。

好啦，到此为止，我们完成了系统调用的实现，并且添加了第一个系统调用 getpid，随着系统的完善，我们还会增加更多系统调用呢，现在总结下增加系统调用的步骤。

（1）在 syscall.h 中的结构 enum SYSCALL_NR 里添加新的子功能号。

（2）在 syscall.c 中增加系统调用的用户接口。

（3）在 syscall-init.c 中定义子功能处理函数并在 syscall_table 中注册。

有关系统调用的内容到此为止，下节中我们将在用户进程中调用 getpid 进行测试。

12.2.7　在用户进程中的系统调用

本节我们将在用户进程中执行系统调用了，想想还是很激动呢。咱们的用户进程还是暂时在 main.c 中用函数来模拟，请见代码 12-9。

代码 12-9　（project/c12/a/kernel/main.c）

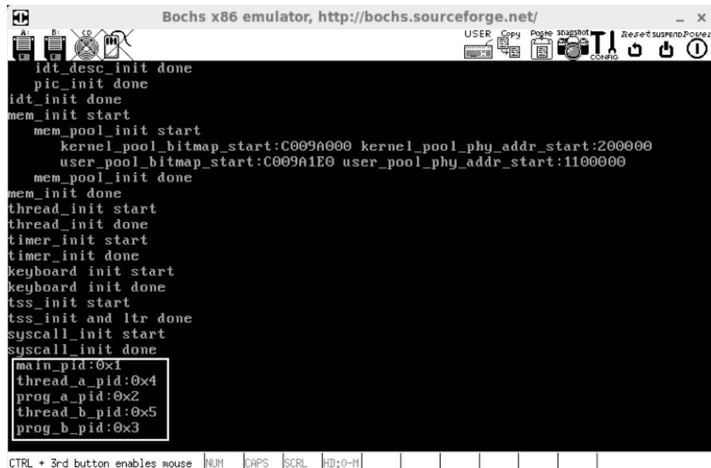
```
…略
 7 #include "syscall-init.h"
 8 #include "syscall.h"
 9
10 void k_thread_a(void*);
11 void k_thread_b(void*);
12 void u_prog_a(void);
13 void u_prog_b(void);
14 int prog_a_pid = 0, prog_b_pid = 0;
15
16 int main(void) {
17     put_str("I am kernel\n");
18     init_all();
19
20     process_execute(u_prog_a, "user_prog_a");
21     process_execute(u_prog_b, "user_prog_b");
22
23     intr_enable();
24     console_put_str(" main_pid:0x");
25     console_put_int(sys_getpid());
26     console_put_char('\n');
27     thread_start("k_thread_a", 31, k_thread_a, "argA ");
28     thread_start("k_thread_b", 31, k_thread_b, "argB ");
29     while(1);
30     return 0;
31 }
32
33 /* 在线程中运行的函数 */
34 void k_thread_a(void* arg) {
```

```
35     char* para = arg;
36     console_put_str(" thread_a_pid:0x");
37     console_put_int(sys_getpid());
38     console_put_char('\n');
39     console_put_str(" prog_a_pid:0x");
40     console_put_int(prog_a_pid);
41     console_put_char('\n');
42     while(1);
43 }
44
45 /* 在线程中运行的函数 */
46 void k_thread_b(void* arg) {
47     char* para = arg;
48     console_put_str(" thread_b_pid:0x");
49     console_put_int(sys_getpid());
50     console_put_char('\n');
51     console_put_str(" prog_b_pid:0x");
52     console_put_int(prog_b_pid);
53     console_put_char('\n');
54     while(1);
55 }
56
57 /* 测试用户进程 */
58 void u_prog_a(void) {
59     prog_a_pid = getpid();
60     while(1);
61 }
62
63 /* 测试用户进程 */
64 void u_prog_b(void) {
65     prog_b_pid = getpid();
66     while(1);
67 }
```

大体上说一下，这里创建了两个用户进程，分别是 u_prog_a 和 u_prog_b，它们都调用 getpid()来获得自己的 pid，返回的 pid 分别存储在全局变量 prog_a_pid 和 prog_b_pid 中，又创建了内核线程 k_thread_a 和 k_thread_b，它们也都调用 sys_getpid()来获得自己的 pid。目前尚未实现为用户进程打印字符的系统调用，因此还是老办法，先让内核线程帮着把用户进程 pid 打印出来，也就是在内核线程 k_thread_a 和 k_thread_b 中分别输出变量 prog_a_pid 和 prog_b_pid。

之前我们验证程序时都是在屏幕上循环输出信息，看着确实很乱，为了屏幕清爽一些，这次我们只输出一次。不过有一点点的不同，这次是进程创建在先，线程创建在后，这样做的目的是同步输出，“尽量”使线程在进程之后执行，避免线程执行完时，进程尚未执行，也就是进程没来得及执行 getpid 时线程已经将变量 prog_[a-b]_pid 输出了，这会导致打印的 pid 为 0。当然，这么做也并不总是靠谱，还是要取决于调度和阻塞时机。

编译、写入磁盘，运行效果如图 12-6 所示。



▲图 12-6　系统调用之 getpid

您看图 12-6 中最下面的方框中，从上到下显示了主线程的 pid，其值为 1，线程 thread_a 的 pid 为 0x4，用户进程 prog_a 的 pid 为 0x2，线程 thread_b 的 pid 为 0x5，进程 prog_b 的 pid 为 0x3。根据 pid 的值可以看出，任务创建的顺序和代码顺序是一样的，主线程是最先创建的，因此 pid 为 1，接着是两个用户进程和两个内核线程，至于打印的顺序并不与任务创建顺序一致，这与创建用户进程的步骤较多、实际调度时机和申请锁时的阻塞有关。

好，本节到这就结束了，大家辛苦了。

12.2.8　系统调用之栈传递参数

我们目前的系统调用是通过寄存器来传递参数的，原因和大伙说过了，若用栈传递参数的话，调用者（用户进程）首先得把参数压在 3 特权级的栈中，然后内核将其读出来再压入 0 特权级栈，这涉及到两种栈的读写，故通过寄存器传递参数效率更高。道理虽然容易说，但依然有很多同学好奇如何通过栈传递参数，更直接地说，是想了解如何在内核态下访问用户态的栈空间。好吧，为满足部分同学的好奇心，这里给大伙儿做个演示，故本节与咱们后续工作无关，仅作为选读。

从内核态访问用户态的栈空间，听起来似乎有点“炫”，但其实一点难度都没有，就是有一点点麻烦，要想在内核态下访问用户态的栈空间，关键在于如何得到用户栈的地址。不过好在处理器已经帮咱们埋下了伏笔，当从用户态进入内核态时，由于特权级发生了变化，处理器会自动在内核栈中压入 3 特权级栈的选择子 SS 及栈指针 esp，故我们在中断处理程序中可以从内核栈中把它们再读出来，由于我们把段描述符设置为了平坦模型，即一个段 4GB 大小，所以只要从内核栈中把 eip 读取出来就行了。有了 3 特权级栈的栈顶指针，再添加一定的偏移量，就能获得用户进程传入的参数。

要完成这次尝试，需要改造两个地方，一个是用户空间中的参数传递，另一个是 0x80 中断处理例程中的参数处理，咱们先改进下用户进程的参数传递部分，如代码 12-10 所示。

代码 12-10　（project/c12/a_stack_syscall/lib/user/syscall.c）

```
 1 #include "syscall.h"
 2
 3 /* 无参数的系统调用 */
 4 #define _syscall0(NUMBER) ({                    \
 5    int retval;                                   \
 6    asm volatile (                                \
 7    "pushl %[number]; int $0x80; addl $4, %%esp"\
 8    : "=a" (retval)                               \
 9    : [number] "i" (NUMBER)                       \
10    : "memory"                                  \
11    );                                              \
12    retval;                                       \
13 })
…略
42 /* 三个参数的系统调用 */
43 #define _syscall3(NUMBER, ARG0, ARG1, ARG2) ({        \
44    int retval;                                       \
45    asm volatile (                                     \
46       "pushl %[arg2]; pushl %[arg1]; pushl %[arg0]; "  \
47       "pushl %[number]; int $0x80; addl $16, %%esp"    \
48       : "=a" (retval)                                  \
49       : [number] "i" (NUMBER),                         \
50         [arg0] "g" (ARG0),                             \
51         [arg1] "g" (ARG1),                             \
52         [arg2] "g" (ARG2)                               \
53       : "memory"                                        \
54    );                                                   \
55    retval;                                              \
56 })
57
58 /* 返回当前任务 pid */
59 uint32_t getpid() {
60    return _syscall0(SYS_GETPID);
61 }
```

这里只列出了两个代表性的_syscall，无参数的_syscall0 和 3 个参数的_syscall3，虽然 getpid 中用到的是_syscall0，但还是介绍下稍微复杂一点的_syscall3，_syscall0 与它同理。系统调用需要参数和子功能号，因此用户程序要在执行 int 0x80 前将参数和子功能号压入用户栈，这里约定下，参数先压入栈，子功能号后压入栈，只有提前确定好它们在栈中的次序，相应的 0x80 号中断处理程序才能正确获取到参数及子功能号。

代码第 43 行是宏_syscall3 的定义，在内联汇编中，这里并没有像 Linux 那样用序号占位符，原因是序号占位符不够灵活，需要与约束的次序绑定，因此这里用的是名称占位符，在第 49～52 行，将同名的大写参数转而用小写名称作为占位符。第 46 行按照 c 调用约定，将 3 个参数从右往左依次入栈。第 47 行做了三件事，先压入子功能号，再执行 int 0x80 触发软中断，最后通过 addl $16, %%esp 使栈顶指针+16，跨过栈中的 3 个参数和子功能号。

下面看一下 0x80 号中断处理例程的代码，如代码 12-11 所示。

代码 12-11　（project/c12/a_stack_syscall/kernel/kernel.S）

```
…略
 98 ;;;;;;;;;;;;;;;;   0x80 号中断   ;;;;;;;;;;;;;;;;
 99 [bits 32]
100 extern syscall_table
101 section .text
102 global syscall_handler
103 syscall_handler:
104
105 ; 系统调用传入的参数在用户栈中，此时是内核栈
106 ;1 保存上下文环境
107    push 0          ; 压入 0，使栈中格式统一
108
109    push ds
110    push es
111    push fs
112    push gs
113    pushad          ; PUSHAD 指令压入 32 位寄存器，其入栈顺序是:
114            ; EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI
115
116    push 0x80      ; 此位置压入 0x80 也是为了保持统一的栈格式
117
118 ;2 从内核栈中获取 cpu 自动压入的用户栈指针 esp 的值
119    mov ebx, [esp + 4 + 48 + 4 + 12]
120
121 ;3 再把参数重新压在内核栈中，此时 ebx 是用户栈指针
122 ;  由于此处只压入了三个参数，所以目前系统调用最多支持 3 个参数
123    push dword [ebx + 12]        ; 系统调用的第 3 个参数
124    push dword [ebx + 8]         ; 系统调用的第 2 个参数
125    push dword [ebx + 4]         ; 系统调用的第 1 个参数
126    mov edx, [ebx]               ; 系统调用的子功能号
127
128    ; 编译器会在栈中根据 C 函数声明匹配正确数量的参数
129    call [syscall_table + edx*4]
130    add esp, 12                  ; 跨过上面的三个参数
131
132 ;4 将 call 调用后的返回值存入待当前内核栈中 eax 的位置
133    mov [esp + 8*4], eax
134    jmp intr_exit                ; intr_exit 返回，恢复上下文
```

第 98～116 行同寄存器传参数的版本相同，第 119 行是在内核栈中获取用户空间的栈指针，给大伙儿说明下“[esp + 4 + 48 + 4 + 12]”，这里的 esp 是当前内核栈顶，后面的数字是偏移量，左边第 1 个 4 是上面的 push 0x80 所占的位置，48 是上面 4 个 push 段寄存器操作和 1 个 pushd 压入的 8 个通用寄存器，总共(4+8)×4＝48 字节，第 2 个 4 是上面占位用的错误码 0。最后的 12 是这样的，中断发生后，处理器由低特权进入高特权级，它会把 ss3、esp3、eflag、cs、eip 依次压入栈中，共 20 字节。为访问到栈中的 esp3，需要跨过 eip、cs 和 eflags，所以添加了 12 字节的偏移量，这样 esp + 4 + 48 + 4 + 12 便指向栈中 esp3 的位置，代码“mov ebx, [esp + 4 + 48 + 4 + 12]”在内核栈中 esp3 位置取值，将值写入寄存器 ebx，故 ebx 此时是用户栈顶指针。

第 123～126 行按照咱们之前的约定，以系统调用参数最先压入，子功能号后压入的顺序，从用户栈中获取用户进程传入的系统调用参数及子功能号。

此时 ebx 中的值便是子功能号，在第 129 行通过“call [syscall_table + edx*4]”调用子功能处理函数。

以上就是通过栈传递参数的系统调用版本，运行结果与寄存器传参的版本一致，不再单独贴图。

好啦，本节到此结束，兄弟们再见。

12.3 让用户进程“说话”

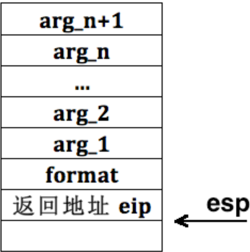
虽然用户进程已经跑起来了，但一直以来我们都借助内核线程帮它打印消息，它从来没真正“开口说话”，这一次我们要为它安装个“嘴巴”，有啥事让它自己“说”，这就是大家非常熟悉的格式化输出函数——printf。

12.3.1　可变参数的原理

在我们平时使用的函数中，大部分都是参数个数已知的函数，比如字符串函数“strlen(const char* s)”，它只有一个参数 s，还有一小部分函数，它们的参数个数是不固定的，这称为可变参数，本节要介绍的 printf 就是其典型应用。

为了引出本节介绍的重点，先得来点铺垫。大伙儿知道，起初人们发明计算机的目的是使人们从重复、冗长、易出错的工作中解脱出来，用程序来代替人工，一方面原因是计算机比人要快，而且在正常情况下不会出错，另一方面是此类工作的流程是固定的，执行过程中不会出现意外的步骤，程序中的每一步都可以提前确定下来，而那时候的操作系统和编译器能力也有限，只支持这种能够提前确定下来的程序，因此那时候的程序也只能够取代人们那些具有重复性、步骤有限且明确的工作。所以，那时的编译器对程序源码的要求很苛刻，不允许源码中存在未知的东西，究其原因，是那时候的编译器和操作系统都较落后，操作系统只会在加载程序时为其分配内存，而且只分配这一次，我们把程序本身占用的内存称为静态内存。而程序在运行时若需要新的内存空间，操作系统就无能为力了，我们把这种程序运行过程中额外需求的内存称为动态内存。由于受限于操作系统，那时候的编译器只支持能够事先确定大小的程序结构，也就是编译出来的程序全部都使用静态内存。在这样艰苦的条件下，人们为了满足运行时的内存要求，在程序源码中预先定义个大数组作为运行时的内存池。随着计算机的进步，操作系统开始支持堆内存管理，堆内存专门用于程序运行时的内存申请，因此编译器也开始支持程序在运行时动态内存申请，也就是编译器开始支持源码中的变长数据结构。程序中的数据结构终归有个长度，此长度要么在编译时确定，要么在运行时确定。编译时确定是指数据结构在源码编译阶段就能确定下来，说白了就是编译器必须提前知道数据结构的长度，它为此类数据结构分配的是静态内存，也就是程序被操作系统加载时分配的内存。运行时确定是指数据结构的长度是在程序运行阶段确定下来的，编译器为此类数据结构（如 C99 中的变长数组）在堆中分配内存，已经说过了，堆本来就是用于程序运行时的动态内存分配，因此可以在运行阶段确定长度。

下面说一下函数。函数占用的也是静态内存，因此也得提前告诉编译器自己占用的内存大小。为了在编译时获取函数调用时所需要的内存空间（这通常是在栈中分配内存单元），编译器要求提供函数声明，声明中描述了函数参数的个数及类型，编译器用它们来计算参数所占据的栈空间。因此编译器不关心函数声明中参数的名称，它只关心参数个数及类型（您懂的，函数声明中的参数可以不包括参数名，但必须包括类型），编译器用这两个信息才能确定为函数在栈中分配的内存大小。重点来了，函数并不是在堆中分配内存，因此它需要提前确定内存空间，这通常取决于参数的个数及类型大小，但编译器却允许函数的参数个数不固定（可变参数），怎么看上去显得那么“动态”？其实可变参数的这种“动态”只是一种幻想，本质上还是静态，这一切得益于编译器采用 C 调用约定来处理函数的传参方式。C 调用约定规定：由调用者把参数以从右向左的顺序压入栈中，并且由调用者清理堆栈中的参数。我们拿格式化输出函数 printf(char* format, arg1, arg2，…)举例，其中的参数 format 就是大伙儿再熟悉不过的包含“%类型字符”的字符串，其调用后栈中布局如图 12-7 所示。



▲图 12-7　可变参数在栈中的布局

您看，既然参数是由调用者压入的，调用者当然知道栈中压入了几个参数，参数占用了多少空间，因此无论函数的参数个数是否固定，采用 C 调用约定，调用者都能完好地回收栈空间，不必担心栈溢出等问题。因此，看似“动态”的可变参数函数，其实也是“静态”“固定”的，传入参数的个数是由编译器在编译阶段就确定下来的。

说了半天，似乎还没说到“点”上，如何知道栈中有多少个参数呢，如何找到它们呢？其实答案全在格式化字符串中，此字符串通常称为 format，不知道什么是格式化字符串？其实您太熟悉了，就是 printf 中的第 1 个参数，比如 printf（”hello %s!”, ”martin”），其中的”hello %s!”便是 format。在格式化字符串中的字符'%'便是在栈中寻找可变参数的依据，紧跟'%'后面的是类型字符，类型字符表示数据类型和进制相关的内容。格式化字符串中有多少'%'，就在栈中找多少次参数，尽管用户（程序员）输入'%'的数量可以和参数个数不一致，但那是用户自己的事，除非用户愿意搬起石头砸自己的脚，编译器也不会检查它们的数量是否匹配，因为参数处理与否是函数自己的行为，由函数体内的代码决定。总之正常情况下，用户传入可变参数的数量应与 format 中字符'%'的数量匹配，以 format 中的'%'作为参数的线索，每找到一个'%'，就到栈中去找一次参数。

好啦，经过上面的讨论，我相信大伙儿已经非常清楚可变参数是怎么回事了，下面说说 Linux 中对可变参数的支持。为方便引用函数中的可变参数，编译器 gcc 的头文件 stdarg.h 中定义了 3 个宏（此文件在我机器上的路径是“/usr/lib/gcc/i686-redhat-Linux/4.4.4/include/”），如图 12-8 所示。

```
47 #define va_start(v,l)    __builtin_va_start(v,l)
48 #define va_end(v)        __builtin_va_end(v)
49 #define va_arg(v,l)      __builtin_va_arg(v,l)
```

▲图 12-8　可变参数宏

这 3 个宏 va_start、va_end 和 va_arg 都以 va（Variable Argument）开头，表示可变参数，但这里它们的值都是以_builtin 为开头的内建符号，就拿 va_start(v,l)来说，它的值为_builtin_va_start(v,l)，那_builtin_va_start 的实现又是什么呢？哈哈，都说了是内建符号了，内建 builtin 的意思就是指在程序内部实现的功能，因此肯定要挖到 gcc 的源码中了，gcc 的内建函数都放在其源码文件 builtins.c 中，拿 gcc-4.9.0 来说，在其源码文件 gcc-4.9.0/gcc/builtins.c 中是用函数 static rtxexpand_builtin_va_start (tree exp)来处理__builtin_va_start 的，具体实现我也不懂了，对编译原理及 gcc 源码有了解的同学可以贡献一下自己的力量，在此小弟先谢过。

虽然无法渗透到编译器中了解这 3 个宏的最终实现，但咱们只要理解这 3 个宏的原理，自己就可以实现一套处理变参的方法。为了弄清楚这几个宏的作用，先在 Linux 中查看帮助，执行 man 3 stdarg 后回车，输出如图 12-9 所示。

```
       #include <stdarg.h>

       void va_start(va_list ap, last);
       type va_arg(va_list ap, type);
       void va_end(va_list ap);
       void va_copy(va_list dest, va_list src);

DESCRIPTION
       A  function  may  be  called  with  a varying number of arguments of varying types.  The include file <stdarg.h> declares a type
       va_list and defines three macros for stepping through a list of arguments whose number and types are not  known  to  the  called
       function.

       The called function must declare an object of type va_list which is used by the macros va_start(), va_arg(), and va_end().

   va_start()
       The va_start() macro initializes ap for subsequent use by va_arg() and va_end(), and must be called first.
  ...略
   va_arg()
       The  va_arg()  macro  expands to an expression that has the type and value of the next argument in the call.  The argument ap is
       the va_list ap initialized by va_start().  Each call to va_arg() modifies ap so that the next call returns  the  next  argument.
       The  argument  type  is  a  type  name  specified  so that the type of a pointer to an object that has the specified type can be
       obtained simply by adding a * to type.
  ...略
   va_end()
       Each invocation of va_start() must be matched by a corresponding invocation of va_end() in the same function.   After  the  call
       va_end(ap)  the variable ap is undefined.  Multiple traversals of the list, each bracketed by va_start() and va_end() are possi-
       ble.   va_end() may be a macro or a function.
```

▲图 12-9　stdarg 帮助

man 3 stdarg 输出的内容还是很多的，图 12-9 只是重点部分的截屏，现在简要介绍下这 3 个宏，为方便介绍它们，和大伙儿约定下：这里的可变参数是指已被压入栈中的 1 个或多个参数，参数个数未知。

ap（argument pointer）是个指针变量，表示参数的指针，用来指向可变参数在栈中的地址。

ap 的类型为 va_list，va_list 是什么呢？大伙儿已经知道 ap 是个指针变量了，故 va_list 本质上是指针

类型，由于 ap 用于指向栈中可变参数的地址，其所指向的参数类型未知，故 va_list 应该是较通用的指针类型，是 void*或 char*都可以，但从名称上看 va_list 是可变参数的列表，这让人联想到字符串 format 中一系列的参数列表“%x%d%f…”，故 va_list 的类型是 char*。

下面是 3 个宏的说明。

（1）va_start(ap,v)，参数 ap 是用于指向可变参数的指针变量，参数 v 是支持可变参数的函数的第 1 个参数（如对于 printf 来说，参数 v 就是字符串 format）。此宏的功能是使指针 ap 指向 v 的地址，它的调用必须先于其他两个宏，相当于初始化 ap 指针的作用。

（2）va_arg(ap,t)，参数 ap 是用于指向可变参数的指针变量，参数 t 是可变参数的类型，此宏的功能是使指针 ap 指向栈中下一个参数的地址并返回其值。

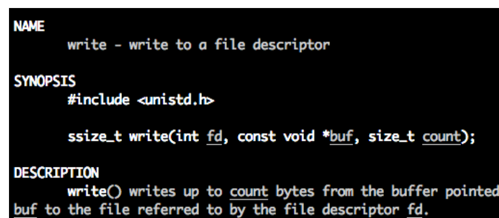
（3）va_end(ap)，将指向可变参数的变量 ap 置为 null，也就是清空指针变量 ap。

好啦，有关可变参数的原理就介绍到这，如果您此时尚未完全明白也没关系，下节咱们会实现以上这 3 个宏，在代码中实践会是理解它们的最佳方式。

12.3.2　实现系统调用 write

我们当初学习 C 语言时，写出的第一句“hello,world\n”就是用 printf 函数来完成的，它是标准 io 函数，因此在使用它之前需要 include <stdio.h>。printf 函数是“格式化”“输出”函数，将格式化后的信息输出到标准输出（通常是屏幕）。但它只是个外壳，真正起到“格式化”作用的是 vsprintf 函数，真正起“输出”作用的是 write 系统调用。您看，白白对 printf 膜拜了 N 多年啊，哈哈，咱们先去实现它的幕后功臣——write 系统调用。查看下 Linux 系统调用 write 的接口，老方法，man 2 write 回车，输出如图 12-10 所示。

write 接受 3 个参数，其中的 fd 是文件描述符，buf 是被输出数据所在的缓冲区，count 是输出的字符数，write 的功能是把 buf 中 count 个字符写到文件描述符 fd 指向的文件中。



▲图 12-10　系统调用 write

您看，由于咱们还没实现文件系统，更谈不上文件描述符 fd 了，故本节完成 write 只能是个简易版，等将来完成文件系统后再把它改造成标准实现。

write 是个系统调用，还记得前面总结的添加系统调用的 3 个步骤吗？咱们按照这 3 个步骤完成简单版 write 系统调用。

第 1 步先在 syscall.h 中的结构 enum SYSCALL_NR 里添加新的子功能号 SYS_WRITE，如代码 12-12 所示。

代码 12-12　（project/c12/b/lib/user/syscall.h）

```
 1 #ifndef __LIB_USER_SYSCALL_H
 2 #define __LIB_USER_SYSCALL_H
 3 #include "stdint.h"
 4 enum SYSCALL_NR {
 5    SYS_GETPID,
 6    SYS_WRITE
 7 };
 8 uint32_t getpid(void);
 9 uint32_t write(char* str);
10 #endif
```

第 2 步在 syscall.c 中增加系统调用的用户接口，如代码 12-13 所示。

代码 12-13　（project/c12/b/lib/user/syscall.c）

```
…略
56 /* 打印字符串 str */
57 uint32_t write(char* str) {
58   return _syscall1(SYS_WRITE, str);
59 }
```

您看，咱们的 write 真的是名符其实的简易版，它只需要一个参数，作用是向屏幕上打印 str。这里用

一个参数的宏_syscall1 来完成系统调用。

第 3 步在 syscall-init.c 中定义子功能处理函数 sys_write 并在 syscall_table 中注册，如代码 12-14 所示。

代码 12-14 （project/c12/b/userprog/syscall-init.c）

```
…略
18 /* 打印字符串 str（未实现文件系统前的版本）*/
19 uint32_t sys_write(char* str) {
20     console_put_str(str);
21     return strlen(str);
22 }
23
24 /* 初始化系统调用 */
25 void syscall_init(void) {
26     put_str("syscall_init start\n");
27     syscall_table[SYS_GETPID] = sys_getpid;
28     syscall_table[SYS_WRITE] = sys_write;
29     put_str("syscall_init done\n");
30 }
```

sys_write 的实现很简单，直接用 console_put_str(str)输出 str。最后用 strlen(str)返回 str 的长度，也就是 write 会返回输出的字符个数。

好啦，write 暂时完成了，目前的简易版对实现 printf 来说已经足够了，待以后有了文件系统后再改进 write。另外，这里就不单独测试 write 啦，以后用 printf 测试也是一样的。

12.3.3 实现 printf

printf 是我们 C 语言标准输出函数，其原型是：

“int printf(const char *format, ...);”

其中的 format 就是格式化字符串，里面包含“%类型字符”，“%类型字符”如“%c”，它用于输出单个字符，“%d”用于输出十进制整型等，“…”表示参数数量不固定，即可变参数。此函数的功能是根据格式化字符串 format 向标准输出打印字符串。如前所述，printf 是 vsprintf 和 write 的封装，write 已经完成，本节要完成 vsprintf、用于可变参数解析的 3 个宏以及转换函数 itoa，这些实现后就完成了基本的 printf，本节的目标是使 printf 支持十六进制输出，即完成“%x”的功能。

先看一下函数 vsprintf 作用，在 Linux 中执行 man vsprintf，可以看到其原型是“int vsprintf(char *str, const char *format, va_list ap);”。此函数的功能是把 ap 指向的可变参数，以字符串格式 format 中的符号'%'为替换标记，不修改原格式字符串 format，将 format 中除“%类型字符”以外的内容复制到 str，把“%类型字符”替换成具体参数后写入 str 中对应“%类型字符”的位置，也就是说函数执行后，str 的内容相当于格式字符串 format 中的“%类型字符”被具体参数替换后的 format 字符串，再次提醒，原格式字符串 format 不做修改，format 只是被参考的格式，如第 1 个参数插入到 str 中对应的第 1 个'%'处，第 2 个参数插入到 str 对应的第 2 个'%'处，vsprintf 执行完成后返回字符串 str 的长度。

您看，之所以 printf 能按照格式化输出，或者说是能把字符串 format 中的“%类型字符”替换为具体参数，这全是 vsprintf 的功劳。咱们马上动手实现它吧，vsprintf 定义在文件 lib/stdio.c 中，请看代码 12-15。

代码 12-15 （project/c12/b/lib/stdio.c）

```
…略
 8 #define va_start(ap, v) ap = (va_list)&v    // 把 ap 指向第一个固定参数 v
 9 #define va_arg(ap, t) *((t*)(ap += 4))      // ap 指向下一个参数并返回其值
10 #define va_end(ap) ap = NULL                // 清除 ap
11
12 /* 将整型转换成字符（integer to ascii） */
13 static void itoa(uint32_t value, char** buf_ptr_addr, uint8_t base) {
14     uint32_t m = value % base;              // 求模，最先掉下来的是最低位
15     uint32_t i = value / base;              // 取整
16     if (i) {                                // 如果倍数不为 0，则递归调用
17         itoa(i, buf_ptr_addr, base);
18     }
19     if (m < 10) {     // 如果余数是 0～9
```

```
20          *((*buf_ptr_addr)++) = m + '0';      //将数字 0～9 转换为字符'0'～'9'
21     } else {          // 否则余数是 A～F
22          *((*buf_ptr_addr)++) = m - 10 + 'A'; //将数字 A～F 转换为字符'A'～'F'
23     }
24 }
25
26 /* 将参数 ap 按照格式 format 输出到字符串 str，并返回替换后 str 长度 */
27 uint32_t vsprintf(char* str, const char* format, va_list ap) {
28     char* buf_ptr = str;
29     const char* index_ptr = format;
30     char index_char = *index_ptr;
31     int32_t arg_int;
32     while(index_char) {
33         if (index_char != '%') {
34              *(buf_ptr++) = index_char;
35              index_char = *(++index_ptr);
36              continue;
37         }
38        index_char = *(++index_ptr);      // 得到%后面的字符
39        switch(index_char) {
40           case 'x':
41              arg_int = va_arg(ap, int);
42              itoa(arg_int, &buf_ptr, 16);
43              index_char = *(++index_ptr);
           // 跳过格式字符并更新 index_char
44              break;
45        }
46     }
47     return strlen(str);
48 }
49
50 /* 格式化输出字符串 format */
51 uint32_t printf(const char* format, ...) {
52     va_list args;
53     va_start(args, format);       // 使 args 指向 format
54     char buf[1024] = {0};         // 用于存储拼接后的字符串
55     vsprintf(buf, format, args);
56     va_end(args);
57     return write(buf);
58 }
```

文件开头定义了用于处理可变参数的 3 个宏，各宏的功能已经和大伙儿介绍过，下面介绍下各宏的实现。其中用到的 va_list 定义在 stdio.h 中，代码是“typedef char* va_list;”，因此 va_list 是字符指针。

va_start(ap, v)的作用是初始化指针 ap，即把 ap 指向栈中可变参数中的第一个参数 v，其实现是 ap = (va_list)&v。ap 和 v 的类型都是 char*，但不同的是 ap 用来存储 v 的地址&v，&v 的类型实际是二级指针，因此在&v 前用(va_list)强制转换为一级指针后再赋值给 ap。

va_arg(ap, t)的作用是使指针 ap 指向栈中下一个参数，并根据下一个参数的类型 t 返回下一个参数的值（啰嗦并严谨着），其实现是*((t*)(ap += 4))。va_arg(ap, t)必须在 va_start(ap, v)之后调用，否则指针 ap 未初始化将导致错误。经 va_start 初始化后，ap 已经指向了栈中可变参数中的第 1 个参数，由于 32 位栈的存储单元是 4 字节，故(ap+=4)将指向下一个参数在栈中的地址，而后将其强制转换成 t 型指针(t*)，最后再用*号取值，即*((t*)(ap += 4))是下一个参数的值。

va_end(ap)的作用就是回收指针 ap，清空，其实现为 ap = NULL。

下面要介绍的函数是 itoa，此函数的作用是将整型转换为字符串，也就是 integer to ascii。其原型是“void itoa(uint32_t value, char** buf_ptr_addr, uint8_t base)”

此函数不属于 Linux，它是 Windows 下的产物，咱们这里借鉴它的功能，取其精华。

虽然此函数并不是今天的主角，但它还是很重要的，itoa 接受 3 个参数，第 1 个参数 value 是待转换的整数，第 2 个参数 buf_ptr_addr 是保存转换结果的缓冲区指针的地址，这里多说两句，缓冲区指针本身已经是指针了，这里说的是指针所在的地址，也是指针的指针，即二级指针，因此类型是 char**。这里用二级指针的原因是：在函数实现中要将转换后的字符写到缓冲区指针指向的缓冲区中的 1 个或多个位置，这取决于进制转换后的数值的位数，比如十六进制 0xd 转换成十进制后变成数值 13，13 要被转换成字符

'1'和'3'，所以数值 13 变成字符后将占用缓冲区中两个字符位置，字符写到哪里是由缓冲区指针决定的，因此每写一个字符到缓冲区后，要更新缓冲区指针的值以使其指向缓冲区中下一个可写入的位置，这种原地修改指针的操作，最方便的是用其下一级指针类型来保存此指针的地址，故将一级指针的地址作为参数传给二级指针 buf_ptr_addr，这样便于原地修改一级指针。如果我没说清楚，一会儿咱们多看看实现中有关递归的代码就清楚了。第 3 个参数 base 是转换的基数，也就是进制。也许您要问了，用这个函数干吗？是这样的，还记得 printf 中的输出类型吗？比如%d 是按照十进制输出，%x 是按照十六进制输出，因此必须要有个数制转换，且将转换结果再转换成字符的函数，这就是 itoa 的使命，下面看它的实现。

iota 的任务有两个：一个是数制转换，原理是把数值对基数求模，先掉下来的是最低位（个位），然后递归调用，依次求出次低位……次高位、最高位，直到数值无法整除基数，也就是没有数位可取，倍数为 0 时结束。这部分功能是由代码第 14～18 行完成的。第 14 行是对基数 base 求模，也就是逐步求出每一级的最低位。第 15 行是对基数 base 取整数倍，此整数倍用于第 17 行的递归调用。另一个是将转换后的数值转换成字符，这部分功能是由代码 19～23 行完成的。在第 19 行，如果掉下来的位 m 是数字 0～9，将其转换成对应字符的 ASCII 码，方法很简单，字符'0'～'9'在 ASCII 码表中是连续的，因此转换原理就是将数字加上字符'0'的 ASCII 码，然后将结果写入缓冲区并更新缓冲区指针（一级指针），即第 20 行的代码“*((*buf_ptr_addr)++) = m + '0'”。在第 21 行，若掉下来的位 m 大于 9（默认为 0xA～0xF，处理较粗糙），则用 0xA～0xF 减去 10（0xA）所得到的差，加上字符 A 的 ASCII 码，便是字符'A'～'F'的 ASCII 码，然后将结果写入缓冲区并更新缓冲区指针（一级指针），即第 22 行的代码“*((*buf_ptr_addr)++) = m - 10 + 'A';”。这里值得注意的是递归调用过程中，第 14 行的求模运算，最先掉下来的是最低位（个位），按照正常的阅读习惯，最高位在左边，最低位在右边，因此最低位虽然是最先得到的，但其转换后的字符却是最后写入缓冲区的，最高位也是一样，虽然最高位是最后得到的，但其对应的字符是最先写入缓冲区的，这涉及到缓冲区指针值的变化，因此函数 itoa 用的是二级指针作为形参，便于原地修改一级指针（此处为缓冲区指针）。试想一下，如果用一级指针作为形参，缓冲区指针作为实参，由于参数是值传递，为更新缓冲区指针的值，要么用个全局变量来记录，要么 itoa 将最新指针返回，多少都有些麻烦，不如二级指针原地修改方便。好啦，再多说就有点喧宾夺主了，毕竟今天的主角不是它，有请主角 vsprintf 登场。

在代码第 27 行是 vsprint 的定义，在本节开头已经把它的原理介绍了，它的功能是将参数 ap 按照格式 format 输出到字符串 str 并返回替换后 str 的长度。

第 28 行用变量 buf_ptr 指向 str，不喜欢对指针型参数直接操作，后续都用 buf_ptr 指代 str。第 29 行用 index_ptr 指代形参 format，此处形参 format 是用 printf 函数中的字符串 format 作为实参代入的。第 30 行 index_char 指向格式字符串 format 中的每个字符，我们用它来找字符'%'。

第 32 行用 while(index_char)循环判断 format 中的每个字符 index_char，直到 index_char 为结束字符'\0'。第 33～37 行用来复制 format 中除'%'以外的字符到 buf_ptr，也就是复制到 str 中。循环遍历中，当 index_char 为字符'%'时，也就是找到了待替换的“%类型字符”，如“%x”为了获取“类型字符”，在第 38 行先使++index_ptr 跳过字符'%'，然后取值，将获取到的类型字符更新 index_char。随后在第 39 行用 switch 结构对 index_char 判断，本节只支持十六进制的输出，也就是类型符号为 x，故 switch 中只有 case 为'x'的分支。在此分支中，通过宏 va_arg(ap, int)获取下一个整型参数，将结果存储到变量 arg_int 中。随后在第 42 行调用 itoa 将 arg_int 转换为十六进制，并存储到 buf_ptr 中，即代码“itoa(arg_int, &buf_ptr, 16);”，注意，这里传入的是 buf_ptr 的地址，前面说过了 itoa 要原地修改 buf_ptr。此时 index_ptr 指向类型字符'x'，故在第 43 行的代码“index_char = *(++index_ptr);”跨过类型字符'x'，更新 index_char 为字符'x'后面的下一个字符，继续下一轮循环在 format 中找字符'%'。

最后要介绍的就是“虚有其表”的 printf 啦，它支持可变参数，因此它的函数声明为 uint32_t printf(const char* format, ...)，其中的“…”表示可变参数。

第 52～53 行定义了变量 args（其实就是 ap），用它来指向参数，并在第 52 行调用宏 va_start(args, format) 对其初始化。

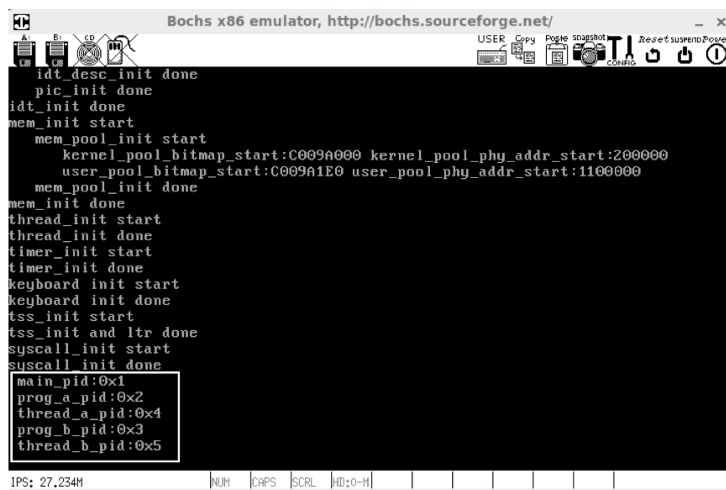
第 54 行定义了 1024 字节大小的数组 buf，用它来存储由 vsprintf 处理的结果，也就是 str，完成之后在第 56 行通过宏 va_end(args)使 args 清空。最后第 57 行执行系统调用 write(buf)，将处理后的字符串输出。

好啦，该说的都说了，到了测试的时候了，还是老样子，修改 main.c，如代码 12-16 所示。

代码 12-16　（project/c12/b/kernel/main.c）

```
…略
 9 #include "stdio.h"
…略
42 /* 在线程中运行的函数 */
43 void k_thread_b(void* arg) {
44     char* para = arg;
45     console_put_str(" thread_b_pid:0x");
46     console_put_int(sys_getpid());
47     console_put_char('\n');
48     while(1);
49 }
50
51 /* 测试用户进程 */
52 void u_prog_a(void) {
53     printf(" prog_a_pid:0x%x\n", getpid());
54     while(1);
55 }
56
57 /* 测试用户进程 */
58 void u_prog_b(void) {
59     printf(" prog_b_pid:0x%x\n", getpid());
60     while(1);
61 }
```

还是以 getpid 为例，这次让用户进程通过 printf 打印出自己的 pid，运行结果如图 12-11 所示。



▲图 12-11　运行的结果

猛地一看，图 12-11 和图 12-6 一样，但其实框框中的打印顺序不一样了，不信您回头对比下，好啦，本节到这就结束了，下节咱们继续丰富 printf 的功能，让它支持更多的输出格式。

12.3.4　完善 printf

上节中，咱们的 printf 版本只支持十六进制“%x”的输出，本节再接再厉，一口气拿下“%c”、“%s”和“%d”。

虽然看上去一下子要完成 3 种类型输出，但由于完成了前期的基础工作，本节的工作量很小。这里还是修改 vsprintf，在其 switch 分支中加入类型字符为 s、c、d 的处理，如代码 12-17 所示。

代码 12-17　（project/c12/c/lib/stdio.c）

```
…略
26 /* 将参数 ap 按照格式 format 输出到字符串 str，并返回替换后 str 长度 */
27 uint32_t vsprintf(char* str, const char* format, va_list ap) {
28     char* buf_ptr = str;
29     const char* index_ptr = format;
```

```
30     char index_char = *index_ptr;
31     int32_t arg_int;
32     char* arg_str;
33     while(index_char) {
34        if (index_char != '%') {
35           *(buf_ptr++) = index_char;
36           index_char = *(++index_ptr);
37           continue;
38        }
39        index_char = *(++index_ptr);      // 得到%后面的字符
40        switch(index_char) {
41           case 's':
42              arg_str = va_arg(ap, char*);
43              strcpy(buf_ptr, arg_str);
44              buf_ptr += strlen(arg_str);
45              index_char = *(++index_ptr);
46              break;
47
48          case 'c':
49              *(buf_ptr++) = va_arg(ap, char);
50              index_char = *(++index_ptr);
51              break;
52
53          case 'd':
54              arg_int = va_arg(ap, int);
55        /* 若是负数，将其转为正数后，在正数前面输出个负号'-' */
56              if (arg_int < 0) {
57                  arg_int = 0 - arg_int;
58                  *buf_ptr++ = '-';
59              }
60              itoa(arg_int, &buf_ptr, 10);
61              index_char = *(++index_ptr);
62              break;
63
64          case 'x':
65              arg_int = va_arg(ap, int);
66              itoa(arg_int, &buf_ptr, 16);
67              index_char = *(++index_ptr);
                // 跳过格式字符并更新 index_char
68              break;
69        }
70     }
71     return strlen(str);
72 }
73
74 /* 同 printf 不同的地方就是字符串不是写到终端，而是写到 buf 中 */
75 uint32_t sprintf(char* buf, const char* format, ...) {
76     va_list args;
77     uint32_t retval;
78     va_start(args, format);
79     retval = vsprintf(buf, format, args);
80     va_end(args);
81     return retval;
82 }
```

本次在代码第 32 行增加了“char* arg_str”，arg_str 是指针变量，专门处理“%s”，也就是打印字符串，在 switch 中增加了对类型字符's'的处理，代码是第 41～46 行。我们知道，当操作数对象为字符串时，编译器只会把字符串的地址作为操作数，并不会把整个字符串中的全部字符复制一份作为参数，这样的好处是显而易见的，节省了时间与空间。在类型字符为's'的分支中，第 42 行通过宏调用 va_arg(ap, char*)获取了待打印字符串的地址，返回给变量 arg_str，第 43 行通过“strcpy(buf_ptr, arg_str)”将待打印字符串 arg_str 拷贝到 buf_ptr 中，这就完成了拼接，然后更新 buf_ptr 的指针，跨过待打印字符串的长度，即代码第 44 行的“buf_ptr += strlen(arg_str)”的作用。第 45 行更新 index_char 的值为类型字符's'后面的字符，即“index_char = *(++index_ptr);”。至此“%s”的处理就完成了，是不是很简单，还有更简单的。

再看单个字符“%c”的处理，第 49 行代码“*(buf_ptr++) = va_arg(ap, char)”直接获得单个字符后写入 buf_ptr，处理更省事，第 50 行同样是使指针 index_ptr 跨过字符'c'，并更新 index_char。“%c”处理结束，

没法再简单了。

下面是十进制整数“%d”的处理，“%d”可用于输出正负整数，关于正负数在计算机中的表现形式还是有些令人费解的，咱们这里要小讨论一下。

人类社会中整数大体上可分为正负两种，计算机为了表示这两类数字，将数字分为无符号数和有符号数。无符号数不关心数字的正负，同人类社会中的自然数一样。对于无符号数来说，它与正负无关，看上去是多少就是多少，显得非常直观。拿 8 位二进制举例，“00000000～11111111”表示的数字用十进制表示是“0～255”，没什么值得怀疑的，无比自然。

有符号数关心数字的正负，但是计算机中只有二进制 0 和 1 两种数字，没有加号'+'和减号'–'，因而无法直接表示数字的正负，只能用间接的方式，所以对于有符号数来说，其表示形式有些“怪异”，还是拿 8 位二进制数举例，正数范围是“00000000～01111111”，用十进制表示是“0～127”，似乎看上去也很直观，然而负数范围是“10000000～11111111”，用十进制表示却是“−128～−1”，这显得很不直观（不知道有没有同学觉得应该是“−0～−127”，哈哈，这是错觉，赶紧忘掉吧），这个负数范围是怎样得到的呢？我们知道，正负数互为相反数，其和为 0，因此我们可以用 0 减去正数来得到负数形式，所得的结果就是补码。咱们源码中表示负号的'–'仅仅是个字符，所有源码文件都是文本，因此对编译器来说，源码就是个长长的文本字符串，编译器分析代码时会将字符'–'转换成计算机中的负数形式。一种可能的情况是（我猜的，学习知识本质上就是找到说服自己理解的方式），当编译器发现负号'–'时，它将'–'后面的数字作为正数，用 0 减去该正数，用这个差来表示负数。拿十进制数−1 来说，−1 等于 0 减去 1，因此可以执行下面的二进制减法做转换。

00000000

<u>- 00000001</u>

11111111

即十进制−1 用二进制 11111111 来表示。

问题来了，二进制的 11111111 到底是无符号数 255，还是有符号数−1 呢？这取决于计算机的视角，不过无论它是什么，都不影响结果的正确性。比如对二进制数 11111111 用指令 dec 执行减 1 操作，结果为 11111110，如果把 11111110 看成是无符号数的话，它表示无符号数 254，如果把 11111110 看成有符号数的话，它又表示−2。因此，参与运算的操作数类型要统一，要么都是无符号数，要么都是有符号数。不过话说回来了，大部分指令对有符号数和无符号数都适配，但还有极少数的指令不能通用处理有符号数和无符号数，因此对于这两种情况需要单独的指令，通常它们在名称上就能区分操作数类型，比如带符号数除法指令 idiv 只用于处理有符号数，而指令 div 只处理无符号数的除法。好啦，咱们继续说代码。

在第 54 行获取了参数后，要判断其值是正还是负，如果是负数的话，咱们要将其转换为相反数，就是第 57 行的代码“arg_int = 0 - arg_int”的用途，然后在其前添加个负号'–'，也就是第 58 行的代码“*buf_ptr++ = '–'”。第 60 行的代码“itoa(arg_int, &buf_ptr, 10)”将数字转换为十进制的字符后写入 buf_ptr。第 61 行还是老样子，更新指针和索引字符。

printf 的相关改进到这就结束了，不过在它的后面连带着把 sprintf 一块加进去了，此函数的功能与 printf 类似，区别是 sprintf 并不把字符串输出到标准输出，而是写到字符串 buf 中，原理也和 printf 一样，不再赘述。

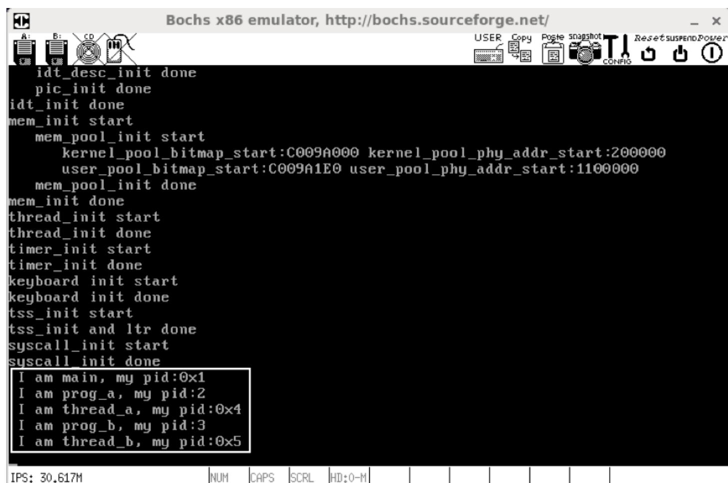
stdio.c 就修改完了，咱们在 main.c 中测试一下，见代码 12-18。

代码 12-18　（project/c12/c/kernel/main.c）

```
…略
42 /* 在线程中运行的函数 */
43 void k_thread_b(void* arg) {
44     char* para = arg;
45     console_put_str(" I am thread_b, my pid:0x");
46     console_put_int(sys_getpid());
47     console_put_char('\n');
48     while(1);
49 }
50
51 /* 测试用户进程 */
52 void u_prog_a(void) {
```

```
53      char* name = "prog_a";
54      printf(" I am %s, my pid:%d%c", name, getpid(),'\n');
55      while(1);
56 }
57
58 /* 测试用户进程 */
59 void u_prog_b(void) {
60      char* name = "prog_b";
61      printf(" I am %s, my pid:%d%c", name, getpid(), '\n');
62      while(1);
63 }
```

这里只更新了两个用户进程中的 printf 函数，每个 printf 都用到了%s、%d、%c 这三种输出格式，运行结果如图 12-12 所示。



▲图 12-12　改进后的 printf

框框中的内容是此次各任务的输出，各字符串输出正常，用户进程 pid 是按照十进制输出的，结尾的换行符也起作用，验证通过。

有关 printf 就到这了，兄弟们下节再会。

12.4　完善堆内存管理

在很久以前就和大伙儿说过，咱们的内存管理系统要大改一次，这事就发生在今天。我们知道，malloc 函数用于程序运行时动态从堆中申请内存，但 malloc 仅仅是堆内存的接口，能够动态分配内存，底层必然有一套完善的内存管理系统在支撑。因此实现 malloc 的前提是咱们先得把底层系统搭起来。这主要涉及在 memory.c 中增加一些数据结构及管理机制，兄弟们，开工啦。

12.4.1　malloc 底层原理

之前我们虽然已经实现了内存管理，但显得过于粗糙，分配的内存都是以 4KB 大小的页框为单位的，当我们仅需要几十字节、几百字节这样的小内存块时，显然无法满足这样的需求了，为此必须实现一种小内存块的管理，可以满足任意内存大小的分配，这就是我们为实现 malloc 要做的基础工作。

这里要引用一个新的名词："arena"，该单词的意思是舞台。arena 是很多开源项目中都会用到的内存管理概念，将一大块内存划分成多个小内存块，每个小内存块之间互不干涉，可以分别管理，这样众多的小内存块就称为 arena。给大伙举个例子，现在的大妈们非常喜欢广场舞，当广场上跳舞的人越来越多时，舞者担心跳舞时肢体会碰到其他舞者而动作放不开，为此，领队给每个舞者画了个大大的方格子，规定每个舞者只允许在自己的方格子里跳，这样大家就不用担心碰到对方，从而跳得很舒展，这个大大的方格子，就是每个舞者的"舞台"。

arena 是由“一大块内存”被划分成无数“小内存块”的内存仓库，我们在原有内存管理系统的基础上实现 arena，大伙儿知道，原有系统只能分配 4KB 粒度的内存页框，因此 arena 的这“一大块内存”也是通过 malloc_page 获得的以 4KB 为粒度的内存，根据请求的内存量的大小，arena 的大小也许是 1 个页框，也许是多个页框，随后再将它们平均拆分成多个小内存块。按内存块的大小，可以划分出多种不同规格的 arena，比如一种 arena 中全是 16 字节大小的内存块，故它只响应 16 字节以内的内存分配，另一种 arena 中全是 32 字节的内存块，故它只响应 32 字节以内的内存分配。我们平时调用 malloc 申请内存时，操作系统返回的地址其实就是某个内存块的起始地址，操作系统会根据 malloc 申请的内存大小来选择不同规格的内存块。因此，为支持多种容量内存块的分配，我们要提前建立好多种不同容量内存块的 arena。

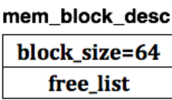
也许 arena 不太好懂，咱们拿面馆举例子，有的人饭量小，有的人饭量大，面馆就为这两类顾客分别准备了小碗面和大碗面（价格当然也就不同了，不过这不重要），我们只要知道有大碗和小碗两种容量的面。面馆的生意特别火，总是有顾客排队等面，煮面的师傅为了大量且同时供应这两种碗面，用了两口大锅同时煮面条，这两口大锅的容量是一样的，煮的面条数量也是一样。面煮熟之后，第 1 口大锅中的面条专用于供应小碗面，它被平均分成 30 份小碗面，第 2 口大锅专用于供应大碗面，它被平均分成 20 份大碗面，这样同时可以满足 30 位买小碗面的顾客和 20 位买大碗面的顾客。这里的两口大锅可以理解为两种不同规格的 arena，第 1 口锅只供应 30 碗小碗面，如同一种只供应 16 字节大小内存块的 arena，第 2 口锅只供应 20 碗大碗面，如同另一种只供应 32 字节大小内存块的 arena，这两种 arena 的总大小都是一口大锅的容量，但由于各自内存块规格容量的不同，两个 arena 各自容纳的内存块数量也是不同的，内存块的数量等于 arena 内存池区域的大小/内存块规格容量。

arena 是个提供内存分配的数据结构，它分为两部分，一部分是元信息，用来描述自己内存池中空闲内存块数量，这其中包括内存块描述符指针（后面介绍），通过它可以间接获知本 arena 所包含内存块的规格大小，此部分占用的空间是固定的，约为 12 字节。另一部分就是内存池区域，这里面有无数的内存块，此部分占用 arena 大量的空间。我们把每个内存块命名为 mem_block，它们是内存分配粒度更细的资源，最终为用户分配的就是这其中的一个内存块。在咱们的实现中，针对小内存块的 arena 占用 1 页框内存，除了元信息外的剩下的内存被平均分成多个小内存块。整个 arena 就像个仓库一样，元信息部分相当于库房管理员，内存块相当于库中物品，如图 12-13 所示。



▲图 12-13　arena 简图

还得继续说面馆的事，大碗面的性价比高，供不应求，一口大锅很难满足供货需要，此时大师傅就会在后厨增加新的大锅来煮面，也就是说提供了多口大锅来为大碗面供货。arena 也是一样的，它容纳的内存块是有限的，总会有内存块供不应求的时候。当某一规格 arena 中的内存块全部分配出去时，必须再增加新的同一规格的 arena，由多个同一规格的 arena 组合为一个“大的仓库”，为同一规格的内存块提供货源。面馆开始卖面时，起初也只是用一口锅供应大碗，根据实际销售情况才增加供货规模。arena 也是一样的，起始为某一类型内存块供货的 arena 只有 1 个，当此 arena 中的全部内存块都被分配完时，系统再创建一个同规格的 arena 继续提供该规格的内存块，当此 arena 又被分配完时，再继续创建出同规格的 arena，arena 规模逐渐增大，逐步形成 arena 集群。既然同一类内存块可以由多个 arena 提供，为了跟踪每个 arena 中的空闲内存块，分别为每一种规格的内存块建立一个内存块描述符，即 mem_block_desc，在其中记录内存块规格大小，以及位于所有同类 arena 中的空闲内存块链表。内存块描述符简图如图 12-14 所示。



▲图 12-14　内存块描述符简图

内存块描述符将所有同类 arena 中空闲内存块汇总，因此它相当于内存块超级大仓库，分配小块内存时必须先经过此入口，系统从它的空闲内存块链表 free_list 中挑选一块内存，也就是说，最终所分配的内存属于此类 arena 集群中某个 arena 的某个内存块。内存块描述符的作用如同面馆的售面窗口，顾客从该窗口就能拿到面，而不用关心这碗面是从哪个大锅里煮出来的。内存块规格有多少种，内存块描述符就有多少种，因此各种内存块描述符的区别就是 block_size 不同，free_list 中指向的内存块规格不同。由于有了内存块描述符，arena 中就没有必要再冗余记录本 arena 中内存块规格信息，而

是用内存块描述符指针指向本 arena 所属的内存块描述符，间接获得本 arena 中内存块的规格大小，内存块描述符指针位于 arena 的元信息当中。

尽管 arena 用小内存块来满足小内存量的分配，但实际上，arena 为内存分配提供了统一的入口，无论申请的内存量是多大，都可以用同一个 arena 来分配内存。小内存块的容量虽然有几种规格，但毕竟是为满足“小”内存量分配的，最大内存块容量不会超过 1024 字节，如果申请的内存量较大，超过 1024 字节，单独的一个小内存块无法满足需求时，这时候您可能想，将多个内存块组合到一起，肯定能满足需求，团结力量大嘛。方法虽具有可行性，但还是太麻烦了，动态维护内存块的信息会增加编程复杂性，这似乎有些像 Linux 的 buddy 系统啦。其实咱们的应用很简单，根本用不着那么麻烦，处理大内存请求时也会创建个 arena，但不会再将它拆分成小内存块，而是直接将整块大内存分配出去，确实有些简单粗暴，但很有效。故此类 arena 没有对应的内存块描述符，元信息中的内存块描述符指针为空，其 arena 简图如图 12-15 所示。



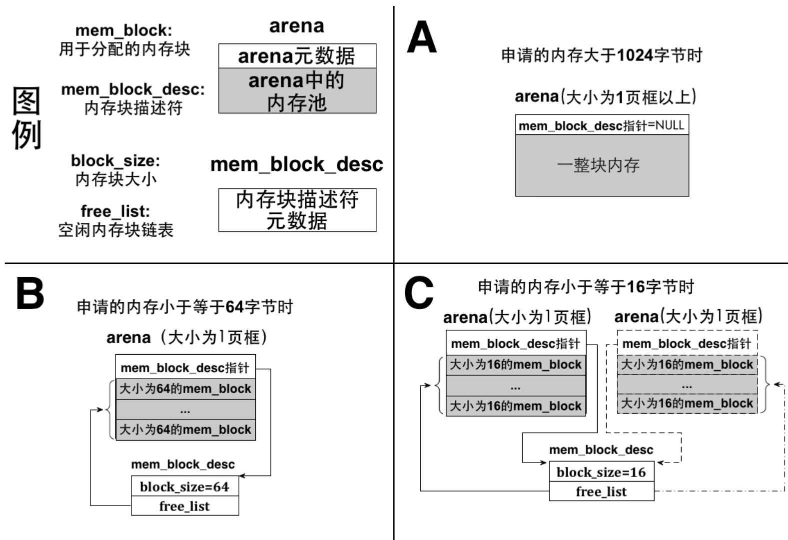
▲图 12-15 大内存 arena 简图

您看，图 12-15 是为了解释大内存时的 arena 结构，图最上面有这样一句，当申请的内存大于 1024 字节时，因此我们对大内存的定义就是大于 1024 字节。为什么要以 1024 为界限呢？有这样一个提前，就是用于处理小内存块时，我们为 arena 分配 1 页框也就是 4KB 大小的内存，我们已经介绍过了，每个 arena 都分为两部分，一部分是占用空间很少的元信息，除元信息外的剩余部分才用于内存块的划分，因此，真正用于内存块的部分不足 4KB。内存块是平均划分的，所以最大的内存块肯定要小于 2KB，这里我们以 2 为底的指数方程来划分内存块，因此最大的内存块是 1024 字节，也就是说对内存块规格为 1024 字节的 arena 来说，它只有 3 个内存块，每个都是 1024 字节，剩余的部分就浪费了。咱们这里的内存块以 16 字节为起始，向上依次是 32 字节、64 字节、128 字节、256 字节、512 字节、1024 字节，总共 7 种规格，因此，内存块描述符也就这 7 种。再次强调一下，每种 arena 中只有一种规格的内存块，并不是同时包含多种规格，比如要么该 arena 中全是 16 字节大小的内存块，要么全是 512 字节的内存块。对于小内存块来说，系统为 arena 分配的内存总共为 4KB，因此，不同规格 arena 中的内存块数量也是不同的，举例来说，假设 arena 元信息大小为 12 字节，对于内存块规格 16 字节的 arena，其包括的内存块数量是(4096-12)/16，对于内存块规格 128 字节的 arena，其包括的内存块数量是(4096-12)/128。

总结，在内存管理系统中，arena 为任意大小内存的分配提供了统一的接口，它既支持 1024 字节以下的小块内存的分配，又支持大于 1024 字节以上的大块内存，malloc 函数实际上就是通过 arena 申请这些内存块。arena 是个内存仓库，并不直接对外提供内存分配，只有内存块描述符才对外提供内存块，内存块描述符将同类 arena 中的空闲内存块汇聚到一起，作为某一规格内存块的分配入口。因此，内存块描述符与 arena 是一对多的关系，每个 arena 都要与唯一的内存块描述符关联起来，多个同一规格的 arena 为同一规格的内存块描述符供应内存块，它们各自的元信息中用内存块描述符指针指向同一个内存块描述符。现在把整个逻辑贯穿起来，如图 12-16 所示。

图 12-16 中左上角是图例，表示了 arena 和 mem_block_desc 元信息及逻辑结构，右上角的 A 图是用于处理大于 1024 字节的大内存的 arena，其大小是 1 页框以上，其中的内存池部分并没有划分成多个小内存块，因此 arena 元信息中，内存块描述符指针 mem_block_desc 值为 NULL。左下角的图 B 是被拆分成 64KB 小内存块的 arena，其指针 mem_block_desc 指向规格为 64 字节的内存块描述符，内存块描述符的空闲内存块链表 free_list 将 arena 中可用内存块汇总。我们说过，当一个 arena 中的内存块不够用时，需要用多个 arena 为同一规格内存块“供货”，图 C 描述的就是这种情况。此例的内存块描述符规格是 16 字节，因此与其关联“供货”的 arena 规格也必须是 16 字节。起初是左边那个 arena 为其提供内存块，当它的内存块分配耗尽时，系统又创建右边的 arena（虚线表示的），从而保证该规格的内存块“货源充足”。

好啦，不知道大伙儿是否理解了 arena，其实只要了解大概意思就行，在后面代码中知道是在做什么就可以了，本节到此结束，下节讨论代码。



▲图 12-16　arena 与 mem_block_desc 的逻辑关系

12.4.2　底层初始化

本节我们要完成上一节中介绍的基础，构建 7 种规格的内存块描述符，这就够了，以后在实际 malloc 时，若发现缺失内存块时再创建相应的 arena，请见代码 12-19。

代码 12-19　（project/c12/d/kernel/memory.h）

```
…略
28 /* 内存块 */
29 struct mem_block {
30     struct list_elem free_elem;
31 };
32
33 /* 内存块描述符 */
34 struct mem_block_desc {
35     uint32_t block_size;          // 内存块大小
36     uint32_t blocks_per_arena;    // 本 arena 中可容纳此 mem_block 的数量
37     struct list free_list;        // 目前可用的 mem_block 链表
38 };
39
40 #define DESC_CNT 7                // 内存块描述符个数
…略
```

在 memory.h 中最先定义的是内存块结构 struct mem_block，由于内存块有 7 个规格，这里提供一种通用结构，将来可以从 arena 中按照内存块规格来拆分。结构中只有一个成员 struct list_elem，用来添加到同规格内存块描述符的 free_list 中。内存块 mem_block 所占用的内存是从 arena 中拆分出来的，其相关属性用 mem_block_desc 来描述。

接下来定义的是内存块描述符结构 struct mem_block_desc，有 3 个成员，free_list 是空闲内存块链表，block_size 是本描述符的规格，它的 free_list 中只能添加规格为 block_size 的内存块。blocks_per_arena 是告诉本 arena 中可容纳规格为 block_size 的内存块的数量。注意，链表 free_list 的长度是无限大的，并不等于 blocks_per_arena 个元素。因为 blocks_per_arena 只是一个 arena 能够提供的内存块个数，而此 free_list 可以由多个 arena 提供内存块。

最后的宏 DESC_CNT 表示内存块描述符的数量，其值为 7，原因是咱们的内存块规格大小是以以 2 为底的指数方程来设计的，从 16 字节起，分别是 16、32、64、128、256、512、1024 字节，共有 7 种规格的内存块。对于小内存的 arena 来说，其大小是一页框 4096 字节，其中的内存块是大小一致的。除了元信息占用的内存外，arena 中不能同时容纳两个 2048 的内存块，2048 下一级内存规格就是 1024，故最大的内存块就是 1024 字节。当申请的内存大小超过 1024 时就直接返回一个页框，不再从 arena 中划分，因为意义不大。

下面再看下最新的 memory.c，如代码 12-20 所示。

代码 12-20　（project/c12/d/kernel/memory.c）

```
…略
 32 /* 内存仓库 */
 33 struct arena {
 34     struct mem_block_desc* desc;  // 此 arena 关联的 mem_block_desc
 35 /* large 为 ture 时，cnt 表示的是页框数。
 36   * 否则 cnt 表示空闲 mem_block 数量 */
 37     uint32_t cnt;
 38     bool large;
 39 };
 40
 41 struct mem_block_desc k_block_descs[DESC_CNT];
                                        // 内核内存块描述符数组
 42 struct pool kernel_pool, user_pool;   // 生成内核内存池和用户内存池
 43 struct virtual_addr kernel_vaddr;     // 此结构用来给内核分配虚拟地址
 44
…略
308 /* 为 malloc 做准备 */
309 void block_desc_init(struct mem_block_desc* desc_array) {
310     uint16_t desc_idx, block_size = 16;
311
312     /* 初始化每个 mem_block_desc 描述符 */
313     for (desc_idx = 0; desc_idx < DESC_CNT; desc_idx++) {
314          desc_array[desc_idx].block_size = block_size;
315
316          /* 初始化 arena 中的内存块数量 */
317          desc_array[desc_idx].blocks_per_arena = \
(PG_SIZE - sizeof(struct arena)) / block_size;
318
319          list_init(&desc_array[desc_idx].free_list);
320
321          block_size *= 2;       // 更新为下一个规格内存块
322     }
323 }
324
325 /* 内存管理部分初始化入口 */
326 void mem_init() {
327     put_str("mem_init start\n");
328     uint32_t mem_bytes_total = (*(uint32_t*)(0xb00));
329     mem_pool_init(mem_bytes_total);       // 初始化内存池
330 /* 初始化 mem_block_desc 数组 descs，为 malloc 做准备 */
331     block_desc_init(k_block_descs);
332     put_str("mem_init done\n");
333 }
```

memory.c 中最先定义的是 arena 结构 struct arena，此结构仅有 3 个成员，占 12 字节的空间，它就是我们所说的 arena 元信息。一直都说 arena 用来提供内存块，这么小的空间怎样提供大量的内存呢？答案是这取决于 arena 的创建方式。将来我们会从堆中创建它，我们会给 arena 结构体指针赋予 1 个页框以上的内存，那时候的 arena 就是个名符其实的内存仓库了，页框中除了此结构体外的部分都将作为 arena 的内存池区域，该区域会被平均拆分成多个规格大小相等的内存块，即 mem_block，这些 mem_block 会被添加到内存块描述符的 free_list。结构中第 1 个成员是 desc，它指向本 arena 中的内存块被关联到哪个内存块描述符，同一规格的 arena 只能关联到同一规格的内存块描述符，比如本 arena 中的内存块规格为 64 字节，desc 只能指向规格为 64 字节的内存块描述符。第 2 个成员是 cnt，它的意义要取决于第 3 个成员 large 的值。当 large 为 ture 时，cnt 表示的是本 arena 占用的页框数，否则 large 为 false 时，cnt 表示本 arena 中还有多少空闲内存块可用，将来释放内存时要用到此项。

接下来定义的是内核内存块描述符数组 k_block_descs[DESC_CNT]，共有 7 种描述符规格。用户进程也有自己的内存块描述符数组，将来定义在 pcb 中。

最后是内存块初始化函数 block_desc_init，此函数接受 1 个参数，内存块描述符数组指针 desc_array，功能是初始化数组内 7 个描述符。前面介绍的内存块描述符就是底层结构，在此主要就是初始化它，其他的 arena 及 arena 中的 mem_block 都是“按需分配”，将来通过 malloc 分配内存块时再创建。这里就是通过

for 循环将 7 种规格的内存块描述符初始化，分别初始化内核内存块描述符的 block_size、blocks_per_arena 和 free_list。block_size 起始值为 16，desc_idx 起始值为 0，循环体的最后会将其乘以 2，因此下标 desc_idx 越低，block_size 越小，也就是说，内核和用户内存块描述符数组中，下标越低的内存块描述符，其表示的内存块容量越小，我们在将来会用到此结论。对 blocks_per_arena 初始化时，减去 arena 的大小后再向下整除，这样保证内存块数量不会越过此 arena 占用的页框边界，不过会浪费一部分内存。最后调用 list_init 初始化内存块描述符的 free_list。

最后在第 331 行把 block_desc_init 添加到 mem_init 中调用。

本节主要是基础部分的构建，还没到具体功能验证的阶段，因此本节到这就结束了，下节再见。

12.4.3　实现 sys_malloc

上节中咱们已经把后台基础设施构建完成，哈哈，其实只是初始化了内存块描述符。也许有同学质疑，内存块是由 arena 提供的，arena 还没有创建，怎么能说基础构建完成了呢？对计算机来说，内存资源再大也不嫌多，它是个宝贵的玩意，不能浪费，必须本着按需分配的原则合理使用，因此内存块并不是提前“盲目”准备好的，它在需要时由程序动态创建，创建它的函数就是 sys_malloc，您懂的，它就是 malloc 对应的子功能处理函数 sys_malloc，sys_malloc 的功能是分配并维护内存块资源，动态创建 arena 以满足内存块的分配，似乎离完成系统调用 malloc 不远了。

为了完成 sys_malloc，咱们这里对 pcb 还做了些小改动，请看代码 12-21。

代码 12-21　（project/c12/e/thread/thread.h）

```
…略
 76 /* 进程或线程的 pcb，程序控制块 */
 77 struct task_struct {
 78     uint32_t* self_kstack;            // 各内核线程都用自己的内核栈
 79     pid_t pid;
…略
 95     uint32_t* pgdir;                  // 进程自己页表的虚拟地址
 96
 97     struct virtual_addr userprog_vaddr;   // 用户进程的虚拟地址
 98     struct mem_block_desc u_block_desc[DESC_CNT];
// 用户进程内存块描述符
…略
101 };
…略
```

为了实现用户进程的堆内存管理，在 pcb 中增加了内存块描述符数组 u_block_desc[DESC_CNT]。另外，该数组在使用前必须要初始化，我们是在 process.c 中的函数 process_execute 中完成初始化工作的，如代码 12-22 所示。

代码 12-22　（project/c12/e/userprog/process.c）

```
…略
 96 /* 创建用户进程 */
 97 void process_execute(void* filename, char* name) {
…略
103     thread->pgdir = create_page_dir();
104     block_desc_init(thread->u_block_desc);
…略
113 }
```

在函数 process_execute 中调用 block_desc_init(thread->u_block_desc)完成了用户内存块描述符数组的初始化工作。好了，下面正式说内存管理方面的改进，请见代码 12-23。

代码 12-23　（project/c12/e/kernel/memory.c）

```
…略
323 /* 返回 arena 中第 idx 个内存块的地址 */
324 static struct mem_block* arena2block(struct arena* a, uint32_t idx) {
325    return (struct mem_block*)\
((uint32_t)a + sizeof(struct arena) + idx * a->desc->block_size);
```

```
326 }
327
328 /* 返回内存块b所在的arena地址 */
329 static struct arena* block2arena(struct mem_block* b) {
330     return (struct arena*)((uint32_t)b & 0xfffff000);
331 }
332
333 /* 在堆中申请size字节内存 */
334 void* sys_malloc(uint32_t size) {
335     enum pool_flags PF;
336     struct pool* mem_pool;
337     uint32_t pool_size;
338     struct mem_block_desc* descs;
339     struct task_struct* cur_thread = running_thread();
340
341 /* 判断用哪个内存池*/
342     if (cur_thread->pgdir == NULL) {     // 若为内核线程
343         PF = PF_KERNEL;
344         pool_size = kernel_pool.pool_size;
345         mem_pool = &kernel_pool;
346         descs = k_block_descs;
347     } else {     // 用户进程pcb中的pgdir会在为其分配页表时创建
348         PF = PF_USER;
349         pool_size = user_pool.pool_size;
350         mem_pool = &user_pool;
351         descs = cur_thread->u_block_desc;
352     }
353
354     /* 若申请的内存不在内存池容量范围内，则直接返回NULL */
355     if (!(size > 0 && size < pool_size)) {
356         return NULL;
357     }
358     struct arena* a;
359     struct mem_block* b;
360     lock_acquire(&mem_pool->lock);
361
362 /* 超过最大内存块1024，就分配页框 */
363     if (size > 1024) {
364         uint32_t page_cnt = \
            DIV_ROUND_UP(size + sizeof(struct arena), PG_SIZE);
            // 向上取整需要的页框数
365
366         a = malloc_page(PF, page_cnt);
367
368         if (a != NULL) {
369            memset(a, 0, page_cnt * PG_SIZE);       // 将分配的内存清0
370
371         /* 对于分配的大块页框，将desc置为NULL，\
            cnt置为页框数，large置为true */
372            a->desc = NULL;
373            a->cnt = page_cnt;
374            a->large = true;
375            lock_release(&mem_pool->lock);
376            return (void*)(a + 1); // 跨过arena大小，把剩下的内存返回
377         } else {
378            lock_release(&mem_pool->lock);
379            return NULL;
380         }
381     } else {     // 若申请的内存小于等于1024
                     // 可在各种规格的mem_block_desc中去适配
382         uint8_t desc_idx;
383
384         /* 从内存块描述符中匹配合适的内存块规格 */
385         for (desc_idx = 0; desc_idx < DESC_CNT; desc_idx++) {
386              if (size <= descs[desc_idx].block_size) {
                 // 从小往大后，找到后退出
387                  break;
388              }
389         }
390
391     /* 若mem_block_desc的free_list中已经没有可用的mem_block，
392      * 就创建新的arena提供mem_block */
```

```
393       if (list_empty(&descs[desc_idx].free_list)) {
394           a = malloc_page(PF, 1);      // 分配 1 页框作为 arena
395           if (a == NULL) {
396               lock_release(&mem_pool->lock);
397               return NULL;
398           }
399           memset(a, 0, PG_SIZE);
400
401      /* 对于分配的小块内存，将 desc 置为相应内存块描述符，
402       * cnt 置为此 arena 可用的内存块数,large 置为 false */
403           a->desc = &descs[desc_idx];
404           a->large = false;
405           a->cnt = descs[desc_idx].blocks_per_arena;
406           uint32_t block_idx;
407
408           enum intr_status old_status = intr_disable();
409
410    /* 开始将 arena 拆分成内存块，并添加到内存块描述符的 free_list 中 */
411           for (block_idx = 0; \
               block_idx < descs[desc_idx].blocks_per_arena; block_idx++) {
412               b = arena2block(a, block_idx);
413               ASSERT(!elem_find(&a->desc->free_list, &b->free_elem));
414               list_append(&a->desc->free_list, &b->free_elem);
415           }
416          intr_set_status(old_status);
417       }
418
419    /* 开始分配内存块 */
420       b = elem2entry(struct mem_block, \
          free_elem, list_pop(&(descs[desc_idx].free_list)));
421       memset(b, 0, descs[desc_idx].block_size);
422
423       a = block2arena(b);  // 获取内存块 b 所在的 arena
424       a->cnt--;           // 将此 arena 中的空闲内存块数减 1
425       lock_release(&mem_pool->lock);
426       return (void*)b;
427    }
428 }
…略
```

代码开头定义了两个 sys_malloc 中会用到的函数，它们是有关 arena 和 mem_block 互相转换的功能。

arena2block 接受两个参数，arena 指针 a 和内存块 mem_block 在 arena 中的索引，函数功能是返回 arena 中第 idx 个内存块的首地址。一会儿我们介绍 sys_malloc 时您就会明白，arena 是从堆中创建的，方法是使 arena 结构体指针指向从堆中返回的一个或多个页框的内存，因此 arena 结构体 struct arena 并不是全部 arena 的大小，结构体中仅有 3 个成员，它就是我们所说的 arena 的元信息。在 arena 指针指向的页框中，除去元信息外的部分才被用于内存块的平均拆分，每个内存块都是相等的大小且连续挨着，因此 arena2block 的原理是在 arena 指针指向的页框中，跳过元信息部分，即 struct arena 的大小，再用 idx 乘以该 arena 中内存块大小，最终的地址便是 arena 中第 idx 个内存块的首地址，最后将其转换成 mem_block 类型后返回。内存块大小记录在由 desc 指向的内存块描述符的 block_size 中。转换过程对应的代码是“return (struct mem_block*)((uint32_t)a + sizeof(struct arena) + idx * a->desc->block_size)”。

第 2 个函数 block2arena 接受一个参数，内存块指针 b。block2arena 用于将 7 种规格的内存块转换为内存块所在的 arena，由于此类内存块所在的 arena 占据 1 个完整的自然页框，所以 arena 中的内存块都属于这 1 页框之内，因此函数原理很简单，内存块的高 20 位地址便是 arena 所在的地址，将此地址转换成(struct arena*)后返回即可，对应代码“return (struct arena*)((uint32_t)b & 0xfffff000)”。

下面要介绍的是今天的主角 sys_malloc，这个函数有点长了，不过原理还是很简单的，收起忐忑的心，不用紧张。

sys_malloc 只有一个参数 size，size 是申请的内存字节数。函数开头定义了一些变量：PF、mem_pool、pool_size 和 descs，它们的值由内存申请者来决定，这里的内存申请者包括内核线程和用户进程两种，第 342～352 行针对这两种情况为它们赋值。

接下来定义了 arena 指针 a 和 mem_block 指针 b，指针 a 用来指向新创建的 arena，指针 b 用来指向 arena 中的 mem_block。如前介绍原理时所述，arena 既可处理大于 1024 字节的大内存分配，也支持 1024 字节以内的小内存分配，各自的实现还是有些区别的，下面要分这两种情况来处理。

首先判断如果申请的内存量大于 1024 字节，先计算内存量 size 需要的页框数，宏 DIV_ROUND_UP 大伙儿已经熟悉了，除法向上取整，计算出的页框数存入变量 page_cnt。下面的代码“a = malloc_page(PF, page_cnt)”就是从堆中创建 arena，也就是把 malloc_page 返回的页框地址赋值给 arena 指针 a。所以，内存中并没有 struct arena 和 struct mem_block 静态实例，只有指向堆中的指针。此时指针 a 指向的内存以页框为粒度，前 12 字节始终是元信息结构。之后调用 memset 将 arena 清 0，其实也可以不清 0，由用户自己的意愿。之后开始初始化 arena 的元信息。对大内存的处理我们直接返回 arena 的内存区就好，不需要再将其拆分成小内存块，因此没有对应的内存块描述符，故“a->desc = NULL”。a->cnt 此时的意义是此 arena 占用的页框数，因此 a->cnt = page_cnt。“a->large = true”表示此 arena 用于处理大于 1024 字节以上的内存分配。arena 中可被用户使用的部分是内存池部分，也就是要跨过 arena 前面的元信息部分，故在第 392 行，用“(a+1)”跨过 arena 元信息，也就是跨过一个 struct arena 的大小。最后通过“return (void*)(a + 1)”把 arena 中的内存池起始地址返回，此地址便是为用户分配的内存地址。

第 381 行处理内存小于 1024 字节的情况。我们有 7 种规格的内存块描述符，把它们都遍历一次，肯定能找到合适的内存块。下面用 for 循环排查所有的内存块描述符，注意啦，我们上节在初始化内存块描述符时，下标越低，其 block_size 的值越小，desc_idx 初始为 0，从低容量的内存块向上找，把 7 种 block_size 都遍历一遍，总有一款内存块最接近 size 字节。比如申请的内存量 size 为 120 字节，规格为 128 字节的内存块是最适合的。

找到后退出循环，desc_idx 便是最合适的内存块索引。在分配之前先要判断是否有可用的内存块，这里是通过内存块描述符中的 free_list 是否为空判断的，具体功能代码是“if (list_empty (&descs[desc_idx].free_list))”。如果 free_list 为空，表示目前的供货商 arena 已经被分配光了，此规格大小的内存块已经没有了，此时需要再创建新的 arena。在第 394 行代码“a = malloc_page(PF, 1)”分配 1 页内存来创建新的 arena，之后用 memset(a, 0, PG_SIZE)清 0。下面为新的 arena 初始化元信息，这次的 arena 用于小内存块分配，所以其 desc 指针必须指向具体的内存块描述符，代码“a->desc = &descs[desc_idx];”使 desc 指向上面找到的内存块描述符。a->large 置为 false，表示此 arena 不用于处理大于 1024 字节的大内存。a->cnt 置为 descs[desc_idx].blocks_per_arena，表示此 arena 现在具有的空闲内存块数量。后面会看到，随着以后的分配，会将 a->cnt 的值减少。

在创建新的 arena 后，下一步是将它拆分成内存块，此部分是在第 411 行的 for 循环开始的，循环次数是 descs[desc_idx].blocks_per_arena，这表示此 arena 将被拆分成的内存块数量。拆分内存块是通过 arena2block 函数完成的，它在 arena 中按照内存块的索引 block_idx 拆分出相应的内存块。指针 b 指向每次新拆分出来的内存块，然后将其添加到内存块描述符的 free_list 中。以后每次发现目标内存块描述符的 free_list 为空时，就重新为这样 block_size 大小的块创建 arena，将 arena 打散成 block_size 大小的内存块，继续添加到内存块描述符的 free_list 中。这样一来，为同一内存块描述符提供内存块的 arena 将越来越多，这些 arena 的 desc 都指向同一个内存块描述符。

下面开始分配内存块，内存块被汇总在内存块描述符的 free_list 中，我们用 list_pop 从 free_list 中弹出一个内存块，此时得到的仅仅是内存块 mem_block 中 list_elem 的地址，因此要用到 elem2entry 宏将其转换成 mem_block 的地址。

第 423 行通过函数 block2arena(b)获取内存块 b 所在的 arena 地址，然后将 a->cnt 减 1，表示空闲内存块少了一个。此项是供将来释放内存使用的，释放内存时会参考 cnt 的值，用来判断是将 mem_block 回收到内存块描述符的 free_list 中，还是直接释放内存块所在的 arena。

第 426 行将内存块 b 的地址转换成 void*后返回，此地址便是用户进程得到的内存地址。

这里总结性地多说两句，在各种 list 中的结点是 list_elem 的地址，并不是 list_elem 所在的“宿主数据结构”（这个词是我自己杜撰的，仅供说清楚问题），比如在就绪队列 thread_ready_list 中的是 pcb 的 general_tag 的地址，pcb 便是 general_tag 的宿主数据结构。宿主数据结构中 list_elem 的地址才是链表中的结点，而 list_elem 中存储的是前驱和后继结点的地址，也就是其他宿主数据结构的 list_elem 的地址。当

结点从链表中脱离时，要将其还原成宿主数据结构才能使用，还原工作是通过宏 elem2entry 完成的，本节的内存块分配便是通过该宏得到内存块的起始地址。内存块地址被返回给用户后，用户可以自由使用此内存块，自然也会把此内存块中的 list_elem 型变量 free_elem 覆盖，不过没关系，它并不影响该内存块的回收和分配，您懂的，free_list 中的元素是 list_elem 的地址，地址是不变的，将来回收或再次分配时依然可以正常使用。

好啦，sys_malloc 现在可用了，咱们该进行功能测试了，还是修改 main.c，见代码 12-24。

代码 12-24　（project/c12/e/kernel/main.c）

```
…略
17 int main(void) {
18     put_str("I am kernel\n");
19     init_all();
20     intr_enable();
21     thread_start("k_thread_a", 31, k_thread_a, "I am thread_a");
22     thread_start("k_thread_b", 31, k_thread_b, "I am thread_b ");
23     while(1);
24     return 0;
25 }
26
27     /* 在线程中运行的函数 */
28     void k_thread_a(void* arg) {
29     char* para = arg;
30     void* addr = sys_malloc(33);
31     console_put_str(" I am thread_a, sys_malloc(33), addr is 0x");
32     console_put_int((int)addr);
33     console_put_char('\n');
34     while(1);
35 }
36
37     /* 在线程中运行的函数 */
38     void k_thread_b(void* arg) {
39     char* para = arg;
40     void* addr = sys_malloc(63);
41     console_put_str(" I am thread_b, sys_malloc(63), addr is 0x");
42     console_put_int((int)addr);
43     console_put_char('\n');
44     while(1);
45 }
```

目前 sys_malloc 是内核级函数，因此这次只添加了两个内核线程来调用它，线程 k_thread_a 中调用 sys_malloc(33)申请 33 字节的内存，然后将返回地址转换成整型后再打印出来，也就是代码“console_put_int((int)addr)”。k_thread_b 的功能类似，只是申请的内存大小是 63 字节。运行结果如图 12-17 所示。



▲图 12-17　sys_malloc 功能测试

图下面框出来的是两个线程的输出，我们注意各线程获得的内存地址，分别是 0xc010200c 和 0xc010204c，下面分析下这两个地址背后的“故事”。这两个线程申请的内存字节一个是 33，一个是 63，它们与规格为

64 字节的内存块最接近，因此 sys_malloc 会创建规格为 64 字节的 arena，然后把它拆分成 64 字节的内存块，由于是第 1 次申请内存且只申请了一种内存块，故系统中只存在这一个 arena。把线程 thread_a 获得的内存地址 0xc010200c 拆分成 0xc0102000+0xc 来看，其中 0xc0102000 是 arena 的首地址，0xc 是 arena 元信息大小，故返回的 0xc010200c 是 arena 中第 1 个 64 字节内存块的地址。线程 thread_b 获得的内存地址是 0xc010204c，它与 0xc010200c 相差为 0x40，即十进制 64 字节，这证明 thread_a 申请的 33 字节也占用了 64 字节的内存块，thread_b 申请的 63 字节占用的是 arena 中第 2 个 64 字节内存块。

目测验证通过，本节到此结束，也许有同学说还没有添加系统调用 malloc 呢，以后咱们连同实现 free 后一块加上吧，赶紧休息吧，IT 人很辛苦，洗洗睡啦。

12.4.4 内存的释放

内存管理系统不仅能分配内存，还应该能回收内存，这是最基本的内存管理机制。一直以来我们对内存的使用都是只借不还，这种情况一直到本节结束。

跟大伙儿一块复习一下：内存的使用情况都是通过位图来管理的，因此，无论内存的分配或释放，本质上都是在设置相关位图中的相应位，都是在读写位图。回收物理地址就是将物理内存池位图中的相应位清 0，无需将该 4KB 物理页框逐字节清 0。回收虚拟地址就是将虚拟内存池位图中的相应位清 0。分配则是相反的，也就是将位图中相应位置为 1 即可。

回想一下，我们分配内存时的一般步骤如下。

（1）在虚拟地址池中分配虚拟地址，相关的函数是 vaddr_get，此函数操作的是内核虚拟内存池位图“kernel_vaddr.vaddr_bitmap”或用户虚拟内存池位图“pcb->userprog_vaddr.vaddr_bitmap”。

（2）在物理内存池中分配物理地址，相关的函数是 palloc，此函数操作的是内核物理内存池位图“kernel_pool->pool_bitmap”或用户物理内存池位图“user_pool->pool_bitmap”。

（3）在页表中完成虚拟地址到物理地址的映射，相关的函数是 page_table_add。

以上三个步骤封装在函数 malloc_page 中。

释放内存是与分配内存相反的过程，咱们对照着设计一套释放内存的方法。

（1）在物理地址池中释放物理页地址，相关的函数是 pfree，操作的位图同 palloc。

（2）在页表中去掉虚拟地址的映射，原理是将虚拟地址对应 pte 的 P 位置 0，相关的函数是 page_table_pte_remove。

（3）在虚拟地址池中释放虚拟地址，相关的函数是 vaddr_remove，操作的位图同 vaddr_get。

我们将以上三个功能封装到函数 mfree_page 中。

也许您对上面的步骤（2）感到好奇，为什么“去除虚拟地址映射”的函数名是 page_table_pte_remove，似乎名称 page_table_remove 与 page_table_add 显得更“门当户对”。原因是这样的，咱们使用的是二级页表，一个虚拟地址在页表中对应两个数据项：页目录项 pde 和页表项 pte。一个 pde 记录一个页表的物理地址，该页表的数据占用 1 个物理页框的空间，1 个 pte 是 4 字节，因此页表中包含 1024 个 pte，每个 pte 记录的是最终与虚拟地址映射的物理页框。如何清除虚拟地址？把整个 pte 清 0？行是行，简单省事，但显得有些“粗暴”，只要把 pte 中的 P 位置为 0 就可以了，该位表示 pte 指向的物理页框的数据是否已在该物理页框中，CPU 只要检测到 P 位为 0，就会认为该 pte 无效，根本不会关心 pte 所指向的物理页框的地址是否属于可访问的物理内存的范围。P 位的实际意义是当可用物理内存较少时，可以将 pte 指向的物理页框中的数据转储到外存上，这样就省出了 4KB 的物理内存空间。将物理页中的数据存储到外存的同时，需要将 pte 的 P 位置为 0。这样在下次访问该 pte 对应的虚拟地址时，由于 pte 的 P 位为 0，CPU 会抛出 pagefault 缺页异常，我们可以在处理 pagefault 异常的中断处理程序中将之前保存到外存的页框数据再次载入到物理内存中，该物理内存可以是原来的物理页，也可以是新的物理页，这取决于实际物理内存的使用情况，然后把目标物理页地址更新到 pte 中，并将 P 位置为 1。pagefault 中断处理程序退出后，CPU 自动会再次访问引起此 pagefault 的虚拟地址，这次发现 pte 的 P 位为 1，从而访问正常，这就是 CPU 原生支持的页式虚拟地址管理策略，话说 Linux 虚拟地址管理也是利用 P 位和 pagefault 异常实现的。总之，只要 pte 的

P 位为 0，CPU 就认为该虚拟地址未做映射，从而达到删除虚拟地址的目的。

按理说，当页表中所有的 pte 都无效时，就可以将页表所在的 4KB 页框回收了，然后在页表所属的 pde 中清除 pde 的 P 位。但是大伙儿知道，1 个页表的容量是 4MB 内存，程序使用的内存一般不超过几十 MB，也就是只使用的几个页表，操作系统想在页表上节省内存，也顶多是回收这几个页表占据的几个物理页框的空间，而且页表中的 pte 很难一下子全部无效，这种情况通常只在程序退出时才出现。再者，进程生命周期往往较短，在这短暂的时间内还要修改页表，似乎得不偿失，即使是那些生命周期较长的守护进程，程序在退出时操作系统也会回收为进程分配的一切资源，这当然包括页表本身占据的空间，因此不值得只为节省一两个页框的内存而在程序运行中频繁操作页表。如上所述，我们对于删除虚拟地址的处理方法仅仅是将 pte 中的 P 位清 0，为了显式说明该函数的特点，命名为 page_table_pte_remove，请大伙儿知晓。

罗哩罗嗦地为个函数名解释了这么多，想必大伙儿也困惑了，我总是担心说得不清楚，因此会贯穿一些周边的内容以求“融会贯通”。按照咱们设计的释放内存的 3 个步骤，请见代码 12-25-1。

代码 12-25-1　（project/c12/f/kernel/memory.c）

```
…略
346 /* 将物理地址 pg_phy_addr 回收到物理内存池 */
347 void pfree(uint32_t pg_phy_addr) {
348     struct pool* mem_pool;
349     uint32_t bit_idx = 0;
350     if (pg_phy_addr >= user_pool.phy_addr_start) {  // 用户物理内存池
351         mem_pool = &user_pool;
352         bit_idx = (pg_phy_addr - user_pool.phy_addr_start) / PG_SIZE;
353     } else {      // 内核物理内存池
354         mem_pool = &kernel_pool;
355         bit_idx = (pg_phy_addr - kernel_pool.phy_addr_start) / PG_SIZE;
356     }
357     bitmap_set(&mem_pool->pool_bitmap, bit_idx, 0); // 将位图中该位清 0
358 }
359
360 /* 去掉页表中虚拟地址 vaddr 的映射，只去掉 vaddr 对应的 pte */
361 static void page_table_pte_remove(uint32_t vaddr) {
362     uint32_t* pte = pte_ptr(vaddr);
363     *pte &= ~PG_P_1;     // 将页表项 pte 的 P 位置 0
364     asm volatile ("invlpg %0"::"m" (vaddr):"memory");    //更新 tlb
365 }
366
367 /* 在虚拟地址池中释放以_vaddr 起始的连续 pg_cnt 个虚拟页地址 */
368 static void vaddr_remove(enum pool_flags pf, \
    void* _vaddr, uint32_t pg_cnt) {
369     uint32_t bit_idx_start = 0, vaddr = (uint32_t)_vaddr, cnt = 0;
370
371     if (pf == PF_KERNEL) {  // 内核虚拟内存池
372         bit_idx_start = (vaddr - kernel_vaddr.vaddr_start) / PG_SIZE;
373         while(cnt < pg_cnt) {
374            bitmap_set(&kernel_vaddr.vaddr_bitmap, \
               bit_idx_start + cnt++, 0);
375         }
376     } else {  // 用户虚拟内存池
377         struct task_struct* cur_thread = running_thread();
378         bit_idx_start =\
            (vaddr - cur_thread->userprog_vaddr.vaddr_start) / PG_SIZE;
379         while(cnt < pg_cnt) {
380            bitmap_set(&cur_thread->userprog_vaddr.vaddr_bitmap, \
               bit_idx_start + cnt++, 0);
381         }
382     }
383 }
…略
```

函数 pfree 接受一个参数，即物理页框地址 pg_phy_addr，功能是将物理页框回收到相应的物理内存池，也就是只回收一个物理页。函数实现还是很简单的，用的都是以前的思路及函数，先根据物理地址池的起始地址判断 pg_phy_addr 属于哪个物理内存池，用变量 mem_pool 指向物理内存池，bit_idx 为物理地址在相应物理内存池中的偏移量，最后通过代码“bitmap_set(&mem_pool->pool_bitmap, bit_idx, 0)”在位图中回收该位。

函数 page_table_pte_remove 接受一个参数，即虚拟地址 vaddr，功能就是如前“醉人”的解释，将 pte 中的 P 位置 0。函数体中，先调用 pte_ptr(vaddr)获取虚拟地址所在的 pte 指针，然后在下一行，通过代码“*pte &=～PG_P_1”使 pte 中的 P 位取反为 0。除此之外，此函数还多做了一件事，就是刷新 tlb。pte 已经被修改，因此咱们要把此 pte 更新到 tlb 中，还记得 tlb 吗？它是页表的高速缓存，俗称快表，tlb 是处理器提供的、用于加速虚拟地址到物理地址的转换过程。当页表发生变化时，tlb 中缓存的数据也应该及时更新，否则程序会运行出错。更新 TLB 有两种方式，一是用 invlpg 指令更新单条虚拟地址条目，另外一个是重新加载 cr3 寄存器，这将直接清空 TLB，相当于更新整个页表。咱们这里只更新了虚拟地址 vaddr 对应的 pte，因此不至于大动干戈针对整个 TLB，咱们采用温柔一点的 invlpg 指令去单独更新 vaddr 对应的缓存。invlpg 的指令格式为“invlpg m”，其中 m 是操作数，表示虚拟地址内存，注意，m 并不是立即数形式的虚拟地址，它必须是实实在在的内存地址的形式。比如更新虚拟地址 0x12345678 的缓存，指令是 invalpg [0x12345678]，而不是 invalpg 0x12345678。invalpg 是汇编指令，因此用下面的一行内联汇编代码“asm volatile ("invlpg %0"::"m" (vaddr):"memory")”来更新虚拟地址 vaddr 在 tlb 缓存中的条目，也就是把页表中 vaddr 所在的 pte 重新写入 tlb。注意，invlpg 的操作数 m 是内存地址，因此位于内联汇编代码输入部中的 vaddr，其约束是内存约束 m。

下一个函数是 vaddr_remove，它接受 3 个参数，pf 是虚拟内存池标志，_vaddr 是待释放的虚拟地址，pg_cnt 是连续的虚拟页框数。函数功能是在虚拟地址池中释放以_vaddr 起始的连续 pg_cnt 个虚拟页地址。函数的实现很简单，虚拟内存地址的管理是在虚拟内存池的位图中，因此先根据 pf 判断是处理哪个虚拟内存池，然后再用位图函数 bitmap_set 将虚拟地址在虚拟内存池位图中相应的位清 0。如果是内核，就针对内核的虚拟内存池 kernel_vaddr 操作，先在第 372 行计算虚拟地址 vaddr 在位图 kernel_vaddr 中的偏移量，存入变量 bit_idx_start 中，然后循环 pg_cnt 次，依次将虚拟内存池位图中的相应位清 0。针对用户虚拟内存池的处理与此同理，只是虚拟内存池位图是当前用户进程 pcb->userprog_vaddr，不再赘述。

下面是代码第二部分，见代码 12-25-2。

代码 12-25-2　（project/c12/f/kernel/memory.c）

```
…略
385 /* 释放以虚拟地址 vaddr 为起始的 cnt 个物理页框 */
386 void mfree_page(enum pool_flags pf, void* _vaddr, uint32_t pg_cnt) {
387     uint32_t pg_phy_addr;
388     uint32_t vaddr = (int32_t)_vaddr, page_cnt = 0;
389     ASSERT(pg_cnt >=1 && vaddr % PG_SIZE == 0);
390     pg_phy_addr = addr_v2p(vaddr);
                                 // 获取虚拟地址 vaddr 对应的物理地址
391
392     /* 确保待释放的物理内存在
        低端 1MB+1KB 大小的页目录+1KB 大小的页表地址范围外 */
393     ASSERT((pg_phy_addr % PG_SIZE) == 0 && pg_phy_addr >= 0x102000);
394
395     /* 判断 pg_phy_addr 属于用户物理内存池还是内核物理内存池 */
396     if (pg_phy_addr >= user_pool.phy_addr_start) {
                                              //位于 user_pool 内存池
397          vaddr -= PG_SIZE;
398          while (page_cnt < pg_cnt) {
399             vaddr += PG_SIZE;
400             pg_phy_addr = addr_v2p(vaddr);
401
402             /* 确保物理地址属于用户物理内存池 */
403             ASSERT((pg_phy_addr % PG_SIZE) == 0 &&\
                pg_phy_addr >= user_pool.phy_addr_start);
404
405             /* 先将对应的物理页框归还到内存池 */
406             pfree(pg_phy_addr);
407
408             /* 再从页表中清除此虚拟地址所在的页表项 pte */
409             page_table_pte_remove(vaddr);
410
411             page_cnt++;
```

```
412         }
413     /* 清空虚拟地址的位图中的相应位 */
414         vaddr_remove(pf, _vaddr, pg_cnt);
415
416     } else {        // 位于 kernel_pool 内存池
417         vaddr -= PG_SIZE;
418         while (page_cnt < pg_cnt) {
419            vaddr += PG_SIZE;
420            pg_phy_addr = addr_v2p(vaddr);
421         /* 确保待释放的物理内存只属于内核物理内存池 */
422            ASSERT((pg_phy_addr % PG_SIZE) == 0 && \
423                  pg_phy_addr >= kernel_pool.phy_addr_start && \
424                  pg_phy_addr < user_pool.phy_addr_start);
425
426            /* 先将对应的物理页框归还到内存池 */
427            pfree(pg_phy_addr);
428
429            /* 再从页表中清除此虚拟地址所在的页表项 pte */
430            page_table_pte_remove(vaddr);
431
432            page_cnt++;
433         }
434     /* 清空虚拟地址的位图中的相应位 */
435         vaddr_remove(pf, _vaddr, pg_cnt);
436     }
437 }
…略
```

此部分代码只有 mfree_page 这一个函数，它接受 3 个参数，pf 是内存池标志，_vaddr 是待释放的虚拟地址，pg_cnt 是连续的页框数，此函数的功能是释放以虚拟地址 vaddr 为起始的 cnt 个物理页框，它是上部分代码 12-25-1 中三个函数的封装。

内存回收工作分为三大步骤，先调用 pfree 清空物理地址位图中的相应位，再调用 page_table_pte_remove 删除页表中此地址的 pte，最后调用 vaddr_remove 清除虚拟地址位图中的相应位。

直接说第 390 行，先调用 addr_v2p(vaddr)获取虚拟地址 vaddr 对应的物理地址，将其保存在 pg_phy_addr 中，然后根据 pg_phy_addr 的值判断它属于内核的物理内存池还是用户物理内存池。内核物理内存池 kernel_pool 的地址位于用户物理内存池 user_pool 的前面，即 kernel_pool 的地址在低地址处，所以可以通过代码“if (pg_phy_addr >= user_pool.phy_addr_start)”判断物理地址 pg_phy_addr 所属的物理内存池。如果 pg_phy_addr 大于 user_pool 的起始物理地址，说明 pg_phy_addr 属于用户内存池。接下来是针对用户物理内存池的处理，通过 while 循环处理 pg_cnt 个页框，每个循环中，都是调用 addr_v2p(vaddr)得到 vaddr 的物理地址 pg_phy_addr，然后调用 pfree(pg_phy_addr)在物理内存池位图中清 0 相应位。再调用 page_table_pte_remove(vaddr)清除 pte，pg_cnt 个页框处理完成后，再调用函数 vaddr_remove(pf, _vaddr, pg_cnt)在虚拟地址池位图中清 0 相应位。后面针对内核内存池的逻辑是一样的，不再赘述。

本节完成了回收内存的基础工作，下一节我们将实现 sys_free，用它来释放内存，您懂的，离系统调用 free 不远了，感谢大伙儿，下节见。

12.4.5　实现 sys_free

我们之前实现的 mfree_page 只能释放页框级别的内存块，这当然不能满足我们的需求，必须支持释放任意字节大小的内存，而这就是 sys_free 的使命。sys_free 是系统调用 free 对应的内核功能函数，因此我们的用户进程马上就能使用 free 函数啦。之前我们所做的基础工作都是为了实现 sys_free，sys_free 基于 mfree_page 和 arena，因此本节并没有新增的内容。

sys_free 是内存释放的统一接口，无论是页框级别的内存和小的内存块，都统一用 sys_free 处理。因此，sys_free 针对这两种内存的处理有各自的方法，对于大内存的处理称之为释放，就是把页框在虚拟内存池和物理内存池的位图中将相应位置 0。对于小内存的处理称之为“回收”，是将 arena 中的内存块重新放回到内存块描述符中的空闲块链表 free_list。好啦，具体还是看代码吧，见代码 12-26。

代码 12-26　（project/c12/g/kernel/memory.c）

```
…略
440 /* 回收内存 ptr */
441 void sys_free(void* ptr) {
442     ASSERT(ptr != NULL);
443     if (ptr != NULL) {
444         enum pool_flags PF;
445         struct pool* mem_pool;
446
447     /* 判断是线程，还是进程 */
448         if (running_thread()->pgdir == NULL) {
449             ASSERT((uint32_t)ptr >= K_HEAP_START);
450             PF = PF_KERNEL;
451             mem_pool = &kernel_pool;
452         } else {
453             PF = PF_USER;
454             mem_pool = &user_pool;
455         }
456
457         lock_acquire(&mem_pool->lock);
458         struct mem_block* b = ptr;
459         struct arena* a = block2arena(b);
            // 把 mem_block 转换成 arena，获取元信息

460         ASSERT(a->large == 0 || a->large == 1);
461         if (a->desc == NULL && a->large == true) { // 大于 1024 的内存
462             mfree_page(PF, a, a->cnt);
463         } else {                        // 小于等于 1024 的内存块
464             /* 先将内存块回收到 free_list */
465             list_append(&a->desc->free_list, &b->free_elem);
466
467             /* 再判断此 arena 中的内存块是否都是空闲，\
                如果是就释放 arena */
468             if (++a->cnt == a->desc->blocks_per_arena) {
469                 uint32_t block_idx;
470                 for (block_idx = 0; \
                    block_idx < a->desc->blocks_per_arena; block_idx++) {
471                     struct mem_block*  b = arena2block(a, block_idx);
472                     ASSERT(elem_find(&a->desc->free_list, &b->free_elem));
473                     list_remove(&b->free_elem);
474                 }
475                 mfree_page(PF, a, 1);
476             }
477         }
478         lock_release(&mem_pool->lock);
479     }
480 }
…略
```

sys_free 只接受 1 个参数，内存指针 ptr，函数功能是释放 ptr 指向的内存。

在函数体前部，判断调用者是内核线程，还是用户进程，并初始化内存池标记 PF 和内存池指针 mem_pool。随后在第 458 行将 ptr 赋值给内存块指针 b，然后通过“struct arena* a = block2arena(b)”获取内存块 b 所在的 arena 指针，此目的是获取 arena 中的元信息，通过元信息中的变量 desc 和 large 的值分别进行下一步处理。

如果 a->desc 的值为 NULL 并且 a->large 的值为 true，这说明待释放的内存（也就是 ptr 指向的内存）并不是在 arena 中的小内存块，而是大于 1024 字节的大内存，其大小是 1 个或多个页框，页框的数量是由 arena 元信息中的变量 cnt 记录的。这是我们在 sys_malloc 中进行内存分配时约定好的，“a->desc 为 NULL，并且 a->large 为 true”表示此 arena 只被用于大内存分配，并不是被拆分成多个内存块，因此不存在此 arena 被多次共享的情况，完全可以释放。如果代码“if (a->desc == NULL && a->large == true)”条件成立的话，就调用“mfree_page(PF, a, a->cnt)”释放 a->cnt 个页框。如果不成立的话，表示待释放的内存是小内存块，流程就进入了第 465 行代码“list_append(&a->desc-> free_list, &b->free_elem)”，将此内存块回收到此内存块描述符的 free_list 中。接下来在第 468 行将“++a->cnt”后的结果与内存块描述符中的 blocks_per_arena 比较，如果相等，这表示此 arena 中的空闲内存块已经达到最大数，说明此 arena 已经没人使用了，可以释放。

于是接下来通过 for 循环，将此 arena 中的所有内存块从内存块描述符的 free_list 中去掉，遍历结束后，通过“mfree_page(PF, a, 1)”释放此 arena。至此，内存回收工作完成。

对于第 465 行的“list_append(&a->desc->free_list, &b->free_elem)”，或许您有些疑惑，心想，刚把内存块 b 放回 free_list，后面若释放 arena 的话，还要将内存块 b 从 free_list 中去掉，似乎多余将它放回链表中，为什么不先判断（a->cnt）是否等于（a->desc->blocks_per_arena-1）呢？如果相等，这说明马上就要释放 arena 了，用不着将内存块 b 放回 free_list。听上去很有道理，这样看似效率高，但转念一想，似乎更麻烦了。您看，我们在 for 循环中遍历此 arena 所有内存块时，用“b =arena2block(a, block_idx)”来获得内存块地址后，再调用 list_remove 从 free_list 中去掉内存块。如果之前未将该内存块放回 free_list，这里在执行 list_remove 前必须跳过该内存块，因此还要判断从 arena2block 返回的地址是否等于 ptr，效率反而更低了。

好了，sys_free 就完成了，我们赶紧在 main.c 中测试一下，main.c 代码如代码 12-27 所示。

代码 12-27　（project/c12/g/kernel/main.c）

```
…略
17 int main(void) {
18      put_str("I am kernel\n");
19      init_all();
20      intr_enable();
21      thread_start("k_thread_a", 31, k_thread_a, "I am thread_a");
22      thread_start("k_thread_b", 31, k_thread_b, "I am thread_b ");
23      while(1);
24      return 0;
25 }
26
27 /* 在线程中运行的函数 */
28 void k_thread_a(void* arg) {
29      char* para = arg;
30      void* addr1;
31      void* addr2;
32      void* addr3;
33      void* addr4;
34      void* addr5;
35      void* addr6;
36      void* addr7;
37      console_put_str(" thread_a start\n");
38      int max = 1000;
39      while (max-- > 0) {
40         int size = 128;
41         addr1 = sys_malloc(size);
42         size *= 2;
43         addr2 = sys_malloc(size);
44         size *= 2;
45         addr3 = sys_malloc(size);
46         sys_free(addr1);
47         addr4 = sys_malloc(size);
48         size *= 2; size *= 2; size *= 2; size *= 2;
49         size *= 2; size *= 2; size *= 2;
50         addr5 = sys_malloc(size);
51         addr6 = sys_malloc(size);
52         sys_free(addr5);
53         size *= 2;
54         addr7 = sys_malloc(size);
55         sys_free(addr6);
56         sys_free(addr7);
57         sys_free(addr2);
58         sys_free(addr3);
59         sys_free(addr4);
60      }
61      console_put_str(" thread_a end\n");
62      while(1);
63 }
64
65 /* 在线程中运行的函数 */
66 void k_thread_b(void* arg) {
67      char* para = arg;
68      void* addr1;
```

```
 69      void* addr2;
 70      void* addr3;
 71      void* addr4;
 72      void* addr5;
 73      void* addr6;
 74      void* addr7;
 75      void* addr8;
 76      void* addr9;
 77      int max = 1000;
 78      console_put_str(" thread_b start\n");
 79      while (max-- > 0) {
 80         int size = 9;
 81         addr1 = sys_malloc(size);
 82         size *= 2;
 83         addr2 = sys_malloc(size);
 84         size *= 2;
 85         sys_free(addr2);
 86         addr3 = sys_malloc(size);
 87         sys_free(addr1);
 88         addr4 = sys_malloc(size);
 89         addr5 = sys_malloc(size);
 90         addr6 = sys_malloc(size);
 91         sys_free(addr5);
 92         size *= 2;
 93         addr7 = sys_malloc(size);
 94         sys_free(addr6);
 95         sys_free(addr7);
 96         sys_free(addr3);
 97         sys_free(addr4);
 98
 99         size *= 2; size *= 2; size *= 2;
100         addr1 = sys_malloc(size);
101         addr2 = sys_malloc(size);
102         addr3 = sys_malloc(size);
103         addr4 = sys_malloc(size);
104         addr5 = sys_malloc(size);
105         addr6 = sys_malloc(size);
106         addr7 = sys_malloc(size);
107         addr8 = sys_malloc(size);
108         addr9 = sys_malloc(size);
109         sys_free(addr1);
110         sys_free(addr2);
111         sys_free(addr3);
112         sys_free(addr4);
113         sys_free(addr5);
114         sys_free(addr6);
115         sys_free(addr7);
116         sys_free(addr8);
117         sys_free(addr9);
118      }
119      console_put_str(" thread_b end\n");
120      while(1);
121 }
```

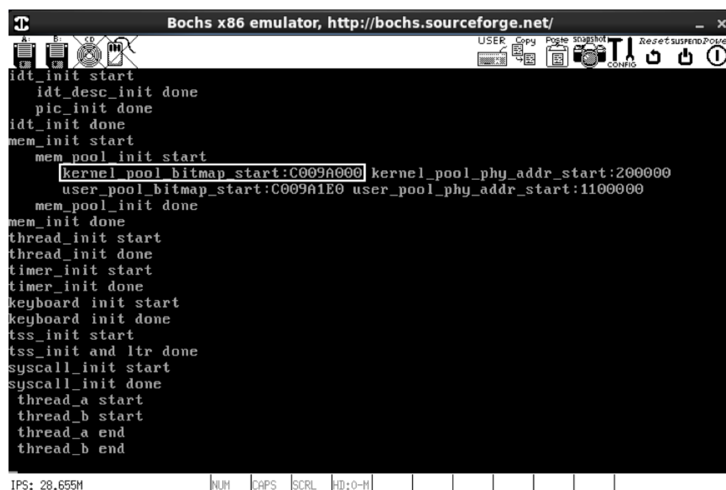
这里创建了两个线程，各线程中分别循环 1000 次，每个循环中多次调用 sys_malloc 和 sys_free，分别申请和释放不同大小的内存。循环开始前，各线程打印出“thread_[ab] start”，当循环结束后，各线程打印出“thread_[ab] end”。相对来说此测试用例还是有些“残酷”的，由于执行的次数较多，故不在屏幕上输出信息了，咱们改为查看内存池位图。

按理说，循环体中 sys_malloc 和 sys_free 是成对匹配的，因此在循环执行前后，位图的情况应该是一致的，这里咱们查看的是内核物理内存池位图，其地址如图 12-18 中的框框所示。

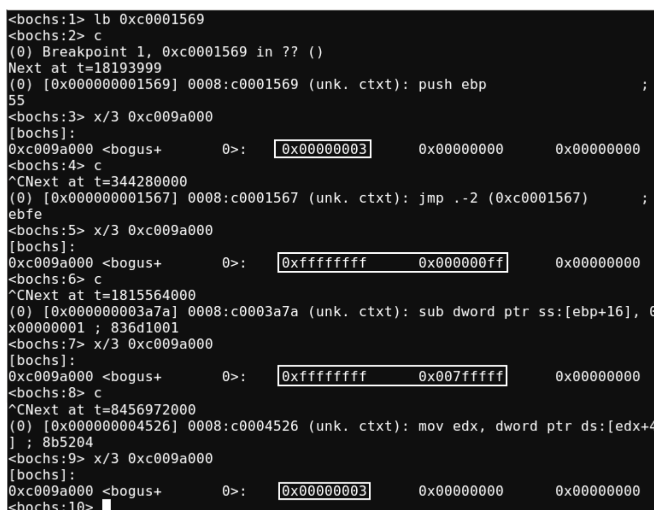
在框框中的是内核物理内存池位图地址，即 0xc009a000。另外，为了对比位图，需要在循环前获得位图的状态，在此以函数 k_thread_a 的地址 0xc0001569 为断点（此地址是用 nm 查看的），此函数先于 k_thread_b 执行，因此可以看到在内存分配与释放之前的原始位图。调试过程如图 12-19 所示。

图 12-14 中第一行执行“lb0xc0001569”设置断点，接着命令 c 连续执行，直到断点处。用“x/10 0xc009a000”查看位图，为了让大伙儿看到位图的变化，我在循环执行过程中按下了“ctrl+c”，又查看了几次位图，

位图的状态都在框框中标出了。当循环执行完成后，也就是在图 12-13 的最下面出现“thread_a end”和“thread_b end”后，图 12-14 中最下面的框框便是位图的最终状态，它与最开始时的值都是 0x00000003，0x3 的二进制是 11，这表示用了 2 个页框，原因是创建两个线程时各用了 1 个页框做其 pcb，因此位图的使用情况与预期相符。



▲图 12-18　内核物理内存池位图地址



▲图 12-19　位图调试

本节到此结束了，下节打算完成 malloc 和 free 的系统调用。

12.4.6　实现系统调用 malloc 和 free

兄弟们，本节的工作很轻松，按照添加系统调用的 3 个步骤，添加 malloc 系统调用和 free 系统调用。在 Linux 中执行 man 3 malloc 回车，屏幕显示 malloc 和 free 的原型是：

“void *malloc(size_t size);”和“void free(void *ptr);”

它们所属的头文件是 stdlib.h。malloc 的功能是分配 size 字节大小的内存，并返回所分配的地址，free 的功能是释放 ptr 所指向的内存。

Linux 中系统调用很多，它们的接口分布在很多头文件中，我也没想过兼容所有 Linux 系统调用接口，为图省事，咱们所有系统调用的接口原型都放在 syscall.h 中。好，现在依照此标准接口实现自己的版本。先在 syscall.h 中添加 malloc 和 free 的系统调用号及接口，如代码 12-28 所示。

代码 12-28 （project/c12/h/lib/user/syscall.h）

```
…略
 4 enum SYSCALL_NR {
 5    SYS_GETPID,
 6    SYS_WRITE,
 7    SYS_MALLOC,
 8    SYS_FREE
 9 };
10 uint32_t getpid(void);
11 uint32_t write(char* str);
12 void* malloc(uint32_t size);
13 void free(void* ptr);
…略
```

接着在 syscall.c 中完成 malloc 和 free 的实现，如代码 12-29 所示。

代码 12-29 （project/c12/h/lib/user/syscall.c）

```
…略
61 /* 申请 size 字节大小的内存，并返回结果 */
62 void* malloc(uint32_t size) {
63   return (void*)_syscall1(SYS_MALLOC, size);
64 }
65
66 /* 释放 ptr 指向的内存 */
67 void free(void* ptr) {
68   _syscall1(SYS_FREE, ptr);
69 }
…略
```

最后在 syscall-init.c 中完成系统调用号与子功能处理函数的关联，也就是更新数组 syscall_table，如代码 12-30 所示。

代码 12-30 （project/c12/h/userprog/syscall-init.c）

```
…略
25 /* 初始化系统调用 */
26 void syscall_init(void) {
27    put_str("syscall_init start\n");
28    syscall_table[SYS_GETPID] = sys_getpid;
29    syscall_table[SYS_WRITE] = sys_write;
30    syscall_table[SYS_MALLOC] = sys_malloc;
31    syscall_table[SYS_FREE] = sys_free;
32    put_str("syscall_init done\n");
33 }
…略
```

到了测试的时候了，在 main.c 中加入相应代码，如代码 12-31 所示。

代码 12-31 （project/c12/h/kernel/main.c）

```
…略
17 int main(void) {
18      put_str("I am kernel\n");
19      init_all();
20      intr_enable();
21      process_execute(u_prog_a, "u_prog_a");
22      process_execute(u_prog_b, "u_prog_b");
23      thread_start("k_thread_a", 31, k_thread_a, "I am thread_a");
24      thread_start("k_thread_b", 31, k_thread_b, "I am thread_b");
25      while(1);
26      return 0;
27 }
28
29 /* 在线程中运行的函数 */
30 void k_thread_a(void* arg) {
31      void* addr1 = sys_malloc(256);
32      void* addr2 = sys_malloc(255);
33      void* addr3 = sys_malloc(254);
34      console_put_str(" thread_a malloc addr:0x");
35      console_put_int((int)addr1);
36      console_put_char(',');
```

```
37      console_put_int((int)addr2);
38      console_put_char(',');
39      console_put_int((int)addr3);
40      console_put_char('\n');
41
42      int cpu_delay = 100000;
43      while(cpu_delay-- > 0);
44      sys_free(addr1);
45      sys_free(addr2);
46      sys_free(addr3);
47      while(1);
48 }
49
50 /* 在线程中运行的函数 */
51 void k_thread_b(void* arg) {
   代码同 k_thread_a
69 }
70
71 /* 测试用户进程 */
72 void u_prog_a(void) {
73      void* addr1 = malloc(256);
74      void* addr2 = malloc(255);
75      void* addr3 = malloc(254);
76      printf(" prog_a malloc addr:0x%x,0x%x,0x%x\n", \
        (int)addr1, (int)addr2, (int)addr3);
77
78      int cpu_delay = 100000;
79      while(cpu_delay-- > 0);
80      free(addr1);
81      free(addr2);
82      free(addr3);
83      while(1);
84 }
85
86 /* 测试用户进程 */
87 void u_prog_b(void) {
   代码同 u_prog_a
99 }
```

本次 main.c 中启了 4 个任务，分别是 2 个用户进程和 2 个内核线程。它们分别都申请了 256、255、254 字节大小的内存，因此它们对应的内存块规格都应该是 256 字节。所有内核线程共享内存空间，因此线程函数 k_thread_a 和 k_thread_b 所申请的内存应该会有地址累加的情况。用户进程拥有独立的内存空间，因此在申请内存时，都会从自己的堆空间从头算起，并不会产生地址累加的情况。256 的十六进制形式是 0x100，比较方便查看地址累加的情况，这就是我们选择规格为 256 字节内存块的原因。各任务中都有代码“while(cpu_delay-- ＞ 0)”，它被插入在内存释放之前，目的是让 CPU 空兜一段时间以延迟释放内存的操作，这样方便大伙儿看到内核线程地址累加的情况。好啦，运行结果如图 12-20 所示。

```
idt_init start
   idt_desc_init done
   pic_init done
idt_init done
mem_init start
   mem_pool_init start
      kernel_pool_bitmap_start:C009A000 kernel_pool_phy_addr_start:200000
      user_pool_bitmap_start:C009A1E0 user_pool_phy_addr_start:1100000
   mem_pool_init done
mem_init done
thread_init start
thread_init done
timer_init start
timer_init done
keyboard init start
keyboard init done
tss_init start
tss_init and ltr done
syscall_init start
syscall_init done
 prog_a malloc addr:0x804800C,0x804810C,0x804820C
 prog_b malloc addr:0x804800C,0x804810C,0x804820C
 thread_a malloc addr:0xC013400C,C013410C,C013420C
 thread_b malloc addr:0xC013430C,C013440C,C013450C
```

▲图 12-20　用户进程和内核线程的内存分配

在图下面的框框中，用户进程 prog_a 三次 malloc 调用，返回的地址分别是 0x804800c、0x804810c、0x804820c，它们各相差 0x100，也就是差 256 字节，用户进程 prog_b 与它相同，原因是用户进程独享内存空间，虚拟地址并不冲突。线程 thread_a 调用 sys_malloc 后返回的地址分别是 0xc013400c、0xc013410c、0xc013420c，地址之间也是相差 0x100，即 256 字节。下面的线程 thread_b 同样调用 sys_malloc，返回的地址出现了累加的现象，在 k_thread_a 地址分配的基础上，从 0xc013430c 开始，然后依次是 0xc013440c 和 0xc013450c，原因是内核线程共享内存空间，虚拟地址必须唯一。

以上测试符合我们的预期，因此有关内存分配的工作暂时各一段落，感谢大伙儿的陪伴，学习的道路上有你们真好。

第 13 章　编写硬盘驱动程序

在很久很久以前，我们在加载 loader.bin 的时候，已经接触了硬盘基本操作。但那时的操作方法还不成体系，咱们为了实现文件系统，必须建立好一套完整的方法，这套方法就是磁盘驱动程序，本章咱们要完成它。

13.1 硬盘及分区表

要实现文件系统，必须先有个磁盘介质，虽然咱们已经有个虚拟磁盘 hd60M.img，但它只充当了启动盘的作用，仅用来存储内核，是个没有文件系统的裸盘（raw disk）。为了省事，这块硬盘咱们就不动了，这里打算把文件系统安装到另一块硬盘上。这一工作涉及到硬盘的一些基础知识，本节给大伙儿介绍下。

13.1.1 创建从盘及获取安装的磁盘数

千里之行始于足下，为编写驱动程序，咱们需要再次从头做起，创建硬盘。hd60M.img 是我们的主盘，上面是咱们的内核。这里咱们再创建一个 80MB 的硬盘作为从盘，在其上安装文件系统。创建硬盘还是利用 bochs 的命令 bximage，此命令在 bochs 安装目录/bin/下，之前已经用过此命令了，这次的创建过程如图 13-1 所示。

```
[work@localhost bochs]$ bin/bximage
========================================================================
                                bximage
                  Disk Image Creation Tool for Bochs
          $Id: bximage.c 11315 2012-08-05 18:13:38Z vruppert $
========================================================================

Do you want to create a floppy disk image or a hard disk image?  选择磁盘类型
Please type hd or fd. [hd] 回车

What kind of image should I create?  选择映像类型，默认是大小固定不变的flat
Please type flat, sparse or growing. [flat] 回车

Enter the hard disk size in megabytes, between 1 and 8257535  以MB为单位的磁盘容量
[10] 80 回车

I will create a 'flat' hard disk image with
  cyl=162     柱面数
  heads=16    磁头数                              磁盘
  sectors per track=63   每磁道扇区数             参数
  total sectors=163296   总共扇区数
  total size=79.73 megabytes  总大小(MB)

What should I name the image?    为映像文件命名
[c.img] hd80M.img 回车

Writing: [] Done.

I wrote 83607552 bytes to hd80M.img.

The following line should appear in your bochsrc:  将下面这行写入bochs配置文件
  ata0-master: type=disk, path="hd80M.img", mode=flat, cylinders=162, heads=16, spt=63
[work@localhost bochs]$
```

▲图 13-1　bochs 创建虚拟硬盘

图上已经对重点内容配上中文说明，带有下画线的部分是键入的内容，方框中的磁盘参数我们在创建分区时会用到。

创建完硬盘，当前目录下便会生成文件 hd80M.img，这就是我们刚刚创建的硬盘。不过图中最下面的一行提示，将下面这行写入 bochsrc，也就是 bochs 的配置文件中。大伙儿要注意了，最下面的这行英文是针对主盘的配置参数，并不是咱们所说的从盘，因此应该把“ata0-master”修改为“ata0-slave”，其他的参数不变。

先不着急把它配置到 bochs 中，咱们得先找个方法证明磁盘确实被安装上了，否则 bochs 未识别的话还不是瞎忙活？在物理地址 0x475 处存储着主机上安装的硬盘的数量，它是由 BIOS 检测并写入的，

咱们本着拿来主义直接用。安装新硬盘之前咱们来查看一下，如图 13-2 所示。

图 13-2 所示为在 bochs 中用 xp 命令查看物理地址 0x475 处的 1 字节，其值是框框中的 0x01，这表示硬盘数是 1，与目前仅有的一块硬盘 hd60M.img 相符。咱们 bochs 的配置文件是 bochsrc.disk，下面将参数写入其中，如图 13-3 所示。

```
<bochs:1> c
^CNext at t=65271420
(0) [0x00000000158f] 0008:c000158f (unk. ctxt):
<bochs:2> xp/b 0x475
[bochs]:
0x00000475 <bogus+       0>:    0x01
<bochs:3>
```

▲图 13-2　bochs 中获取安装的硬盘数

```
33 # 硬盘设置
34 ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
35 ata0-master: type=disk, path="hd60M.img", mode=flat, cylinders=121, heads=16, spt=63
36 ata0-slave: type=disk, path="hd80M.img", mode=flat, cylinders=162, heads=16, spt=63
37
```

▲图 13-3　在配置文件中添加从盘

下面再从 bochs 中验证下是否安装成功了，如图 13-4 所示。

```
<bochs:1> c
^CNext at t=16883678
(0) [0x0000000039ce] 0008:c00039ce (unk. ctxt):
<bochs:2> xp/b 0x475
[bochs]:
0x00000475 <bogus+       0>:    0x02
<bochs:3>
```

▲图 13-4　bochs 中获取安装的硬盘数

果然，框框中的数字现在是 0x02，这说明安装成功了，首战告捷，下一步准备为它创建分区，下节再见。

13.1.2　创建磁盘分区表

什么是文件系统？曾经有人这样告诉我：“给磁盘创建文件系统相当于在空白纸上打好格子，这样在小格子中书写文字就不会乱了”。因此，小弟曾一度误以为文件系统仅仅是有关文件存储格式的静态数据结构，其实不然。

文件系统是运行在操作系统中的软件模块，是操作系统提供的一套管理磁盘文件读写的方法和数据组织、存储形式，因此，文件系统=数据结构+算法，哈哈，所以它是程序。它的管理对象是文件，管辖范围是分区，因此它建立在分区的基础上，每个分区都可以有不同的文件系统。但咱们刚创建了磁盘而已，磁盘还是裸盘，即传说中的 raw disk，本节的任务是把刚创建的磁盘 hd80M.img 分区。这里用的是 fdisk 工具，在分区过程中会用到图 13-1 中的磁盘参数。为了让大伙儿理解 fdisk 的分区过程，咱们先从物理结构上理解磁盘，以下内容可以参考图 3-28，没错，就是很久很久以前的那张机械式硬盘示意图。

盘片：类似光盘中的一个圆盘，上面布满了磁性介质。

扇区：扇区是硬盘读写的基本单位，它在磁道上均匀分布，与磁头和磁道不同，扇区从 1 开始编号。扇区的大小字节数=256×N，N 为自然数。通常取 N 为 2，因此扇区大小为 512 字节。也许有读者会说，我怎么听说的文件存储都是以簇或块为单位的？“簇”或“块”这些是操作系统中读写数据的单位，并不是磁盘原生支持的，一个簇或块等于 1 个以上的扇区。因为磁盘本身就是整个机器的瓶颈，它是速度较低的设备，若操作系统总去访问这些低速设备就太浪费时间了，因此操作系统不可能一次只写一个扇区，为了优化 I/O，操作系统把数据积攒到“多个扇区”时再一次性写入磁盘，这里的“多个扇区”就是指操作系统的簇或块。通常标准库函数还进行了二次优化，数据可以积攒到多个族或块时才写入，不过标准库中还提供了控制选项，可以立即把数据刷进硬盘。

磁道：盘片上的一个个同心圈就是磁道，它是扇区的载体，每一个磁道由外向里从 0 开始编号。同一盘片上的每一个磁道上都由扇区组成，即磁道其实是一圈扇区。磁盘上的磁道数取决于制作工艺。有的同学又说了，离圆心近的磁道与最外圈的磁道周长肯定不一致，那这两种磁道上的扇区数一样吗？答案是：老硬盘是一样的，新式硬盘中已经改进了，外圈磁道会容纳更多的扇区，在新硬盘中有个地址转换器来兼容老硬盘的扇区寻址，因此咱们依然可以认为硬盘中每个磁道上的扇区数一样多。

磁头：就是磁头，哈哈，可以粗略理解为磁带中的磁头。毕竟需要某个设备来读盘片上的数据，这个设

备就是磁头。一个盘片分为上下两个面，各面都有一个磁头，因此一个盘片包括两个磁头，磁头号就表示盘面，平时所说的盘面号就是磁头号。虽然单个盘片的容量不断在增长，但其潜力毕竟是有限的。为了实现大容量，硬盘中必须由多个盘片来组成。既然有多个盘片，两个磁头就自然不够用了，肯定要有盘片×2 个磁头，磁头编号由上到下从 0 开始。

柱面：硬盘是整个计算机系统中很大的瓶颈了，如何才能让硬盘的读写更快，工程师们想到了并行写入的方式。这个并行就是指多个磁头同时写入。也就是通常我们在写一个文件时，是由多个磁头同时写入到不同的盘面中编号位置相同的磁道上，采用并行的方式，读写速度是单盘的（磁头数）倍。这些由不同盘面上的编号相同的磁道（这些编号相同的同心圆大小一致）从上到下所组成的圆柱体的回转面就称为柱面，因此柱面的大小等于盘面数（磁头数）乘以每磁道扇区数。既然一组编号相同的磁道是柱面，而且柱面上的所有磁道号都相同，所以磁道号就称为柱面号。

分区：是由多个编号连续的柱面组成的，因此分区在物理上的表现是由某段范围内的所有柱面组成的通心环，并不是像“饼图”那种逻辑表示，当然若整个硬盘只有 1 个分区，那这个分区就是个所有柱面组成的圆柱体。分区不能跨柱面，也就是同一个柱面不能包含两个分区，一个柱面只属于一个分区，分区的起始和终止都落在完整的柱面上，并不会出现多个分区共享同一柱面的情况，这就是所谓的“分区粒度”。因此，分区大小等于“每柱面上的扇区数”乘以“柱面数”，这就是我们实际分区时，键入的大小往往与实际大小不同的原因，分区大小总会是“每柱面上的扇区数”的整数倍，也就是会以柱面向上取整。假如硬盘上有 n 个柱面，1～10 柱面是 a 分区，11～23 是 b 分区，这两个分区不共享 11 号柱面。

介绍这些就够用了，现在得出公式。

（1）硬盘容量=单片容量×磁头数。

（2）单片容量=每磁道扇区数×磁道数×512 字节。

磁道数又等于柱面数，因此将公式 2 代入公式 1 后：

硬盘容量=每磁道扇区数×柱面数×512 字节×磁头数

大伙再回头看下图 13-1 白色框框中的磁盘参数，柱面数=162，磁头数=16，每磁道扇区数=63，总大小是 79.73MB。这是工具 bximage 为咱们配置好的硬件参数，将它们代入公式 3，硬盘容量=63×162×512×16=83607552 字节，将其换算成 MB 为 79.734375MB，约等于 79.73MB。验证下 hd80M.img 的真实大小，如图 13-5 所示。

```
[work@localhost bochs]$ ll hd80M.img
-rw-rw-r--. 1 work work 83607552 5月    2 07:54 hd80M.img
```

▲图 13-5　hd80M.img 字节大小

图中框框中的 83607552 便是 hd80M.img 的字节大小，与计算出来的大小相符。

一般情况下，每磁道扇区数都是 63，扇区大小都是 512，柱面数和磁头数取决于实际配置的。因此，在硬盘容量已知的情况下：柱面数×磁头数=硬盘容量/63/512，我们只要凑出合适的柱面数和磁头数就行了。比如本例中，柱面数×磁头数=83607552/63/512=2592.000000，因此以下的柱面数和磁头数都可以满足，如“162 和 16”，“324 和 8”，“216 和 12”，“144 和 18”等，不再列举。

也许您在想，介绍这些干吗？是这样的，我们马上要使用的 fdisk 工具，在分区过程中需要设置柱面数和磁头数，如果设置得不合适，这将改变虚拟硬盘的大小。所以，尽管 hd80M.img 已经制造出来了，似乎很“稳定”，但它是用软件虚拟的，并非物理硬盘，fdisk 会根据实际参数将其尺寸“扩张”。在未分区前它是 80MB 大小，如果在 fdisk 分区过程中磁头数和柱面数设置不合适，hd80M.img 就会按照公式 3 变成新的尺寸。因此，我们下面使用 fdisk 时，最好按照 bximage 配置的硬件参数来设置。

按理说咱们应该用 fdisk 分区了，但不知道大伙儿是否对分区都熟悉，为了分区过程顺利进行，还得介绍点分区的基本知识，这里本着能顺利完成 fdisk 就行，下节咱们再对照实际的分区详细说明。

当初硬盘制造者认为，一台机器上顶多安装 4 个操作系统，每个操作系统各占 1 个分区，所以硬盘支持 4 个分区足矣。想想也是，谁没事在 1 台电脑上不断重启机器只为来回切换 4 个操作系统呢？即使现在这么做的人也不多（我同时装过 3 个系统）。可那时限于制造工艺，硬盘容量比较小，用 4 个分区管理小硬盘也确实够用了（顶多几 MB 大小的硬盘，分成多个分区也意义不大），随着硬盘容量越来越大，为方便文件管理，必须想办法支持更多的分区。

分区是逻辑上划分磁盘空间的方式，归根结底是人为地将硬盘上的柱面扇区划分成不同的分组，每个分组都是单独的分区。各分区都有“描述符”来描述分区本身所在硬盘上的起止界限等信息，在硬盘的MBR中有个64字节“固定大小”的数据结构，这就是著名的分区表，分区表中的每个表项就是一个分区的“描述符”，表项大小是16字节，因此64字节的分区表总共可容纳4个表项，这就是为什么硬盘仅支持4个分区的原因。也许有同学会说，把分区表长度定义的大一些，只要能够容纳更多的表项，不就是支持更多的分区了吗？道理确实是这样，但别忘了，任何事物的发展都离不开兼容，在支持更多的分区的同时也必须要兼容现有的这套分区表方案。其实分区表的长度并不是由结构本身限制的，而是由其所在的位置限制的，它必须存在于MBR引导扇区或EBR引导扇区中。在这512字节中，前446字节是硬盘的参数和引导程序，然后才是64字节的分区表，最后是2字节的魔数55aa。随着计算机天长地久的发展，还真别小看这个扇区，很多程序已经对它有依赖了，尤其是一些引导型程序（如BIOS），都会在该扇区的512字节中的固定位置读取关键数据，如果更改了此扇区中的数据结构长度，江湖必然大乱。为此，硬盘厂商准备在分区“描述符”动动手脚。在这个“描述符”中有个属性是文件系统id，它表示文件系统的类型，为支持更多的分区，专门增加一种id属性值（id为5），用来表示该分区可被再次划分出更多的子分区，这就是逻辑分区。因为只是在分区表项中通过属性来判断分区类型，所以这4个分区中的任意一个都可以作为扩展分区。扩展分区是可选项，有没有都行，但最多只有1个，1个扩展分区在理论上可被划分出任意多的子扩展分区，因此1个扩展分区足够了。注意了，这里所说的是理论上支持无限多的划分，由于硬件上的限制，分区数量也变得有限，比如ide硬盘只支持63个分区，scsi硬盘只支持15个分区。硬盘本来没有扩展分区的概念，为了突破4个分区的限制才提出了扩展分区，为了区别这一概念，将剩下的3个区称为主分区。

发明扩展分区的目的是为了支持任意数量的分区，但具体划分出多少分区，完全是由用户决定的，所以，扩展分区是种抽象、不具实体的分区，它类似于一种“宣告”，告诉大家此分区需要再被划分出子分区，也就是所谓的逻辑分区，逻辑分区才可以像其他主分区那样使用。因此，逻辑分区只存在于扩展分区，它属于扩展分区的子集。

综上所述，分区表中共4个分区，哪个做扩展分区都可以，扩展分区是可选的，但最多只有1个，其余的都是主分区。在过去没有扩展分区时，这4个分区都是主分区，为了兼容4个主分区的情况，扩展分区中的第1个逻辑分区的编号从5开始。

一不小心就多说了，我们开始分区吧，过程如图13-6～图13-12所示。

```
[work@localhost bochs]$ fdisk -l ./hd80M.img

Disk ./hd80M.img: 0 MB, 0 bytes
255 heads, 63 sectors/track, 0 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x00000000

[work@localhost bochs]$
[work@localhost bochs]$ fdisk ./hd80M.img
Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel
Building a new DOS disklabel with disk identifier 0xfdf67f62.
Changes will remain in memory only, until you decide to write them.
After that, of course, the previous content won't be recoverable.

Warning: invalid flag 0x0000 of partition table 4 will be corrected by w(rite)
You must set cylinders.
You can do this from the extra functions menu.

WARNING: DOS-compatible mode is deprecated. It's strongly recommended to
         switch off the mode (command 'c') and change display units to
         sectors (command 'u').

Command (m for help): m
Command action
   a   toggle a bootable flag
   b   edit bsd disklabel
   c   toggle the dos compatibility flag
   d   delete a partition
   l   list known partition types
   m   print this menu
   n   add a new partition
   o   create a new empty DOS partition table
   p   print the partition table
   q   quit without saving changes
   s   create a new empty Sun disklabel
   t   change a partition's system id
   u   change display/entry units
   v   verify the partition table
   w   write table to disk and exit
   x   extra functionality (experts only)
```


▲图13-6　fdisk过程1

```
Command (m for help): n
You must set cylinders.
You can do this from the extra functions menu.

Command (m for help): x

Expert command (m for help): m
Command action
   b   move beginning of data in a partition
   c   change number of cylinders
   d   print the raw data in the partition table
   e   list extended partitions
   f   fix partition order
   g   create an IRIX (SGI) partition table
   h   change number of heads
   i   change the disk identifier
   m   print this menu
   p   print the partition table
   q   quit without saving changes
   r   return to main menu
   s   change number of sectors/track
   v   verify the partition table
   w   write table to disk and exit

Expert command (m for help): c
Number of cylinders (1-1048576): 162

Expert command (m for help): h
Number of heads (1-256, default 255): 16

Expert command (m for help): r
```


▲图13-7　fdisk过程2

```
Command (m for help): n                    键入n创建分区
Command action
   e   extended
   p   primary partition (1-4)
p                                          键入p创建主分区
Partition number (1-4): 1                  指定主分区号为1(1~4皆可)
First cylinder (1-162, default 1):         为分区指定起始柱面，使用默认值
Using default value 1
Last cylinder, +cylinders or +size{K,M,G} (1-162, default 162): 32
                                           为分区指定结束柱面号这里设置32
Command (m for help): n
Command action                    键入n创建分区
   e   extended
   p   primary partition (1-4)    键入e创建扩展分区
e
Partition number (1-4): 4         指定扩展分区号为4(2~4皆可)
First cylinder (33-162, default 33): 为分区指定起始柱面号，使用默认值
Using default value 33
Last cylinder, +cylinders or +size{K,M,G} (33-162, default 162):   为分区指定
Using default value 162                                            结束柱面号，
                                                                   使用默认值，
Command (m for help): p   键入p显示目前分区                        全部为扩展分区

Disk ./hd80M.img: 0 MB, 0 bytes
16 heads, 63 sectors/track, 162 cylinders
Units = cylinders of 1008 * 512 = 516096 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0xfdf67f62

       Device Boot      Start         End      Blocks   Id  System
./hd80M.img1                1          32       16096+  83  Linux
./hd80M.img4               33         162       65520    5  Extended   扩展分区Id为5
```

▲图 13-8　fdisk 过程 3

```
Command (m for help): n   键入n继续创建分区
Command action
   l   logical (5 or over)
   p   primary partition (1-4)   键入l创建逻辑分区
l
First cylinder (33-162, default 33):       第1个逻辑分区的柱面范围是33~50
Using default value 33
Last cylinder, +cylinders or +size{K,M,G} (33-162, default 162): 50

Command (m for help): n
Command action
   l   logical (5 or over)
   p   primary partition (1-4)
l
First cylinder (51-162, default 51):       第2个逻辑分区的柱面范围是51~75
Using default value 51
Last cylinder, +cylinders or +size{K,M,G} (51-162, default 162): 75
                                                 略过3个逻辑分区的创建，
Command (m for help): p   键入p显示分区          过程相同

Disk ./hd80M.img: 0 MB, 0 bytes
16 heads, 63 sectors/track, 162 cylinders
Units = cylinders of 1008 * 512 = 516096 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0xfdf67f62

       Device Boot      Start         End      Blocks   Id  System
./hd80M.img1                1          32       16096+  83  Linux
./hd80M.img4               33         162       65520    5  Extended
./hd80M.img5               33          50        9040+  83  Linux
./hd80M.img6               51          75       12568+  83  Linux
./hd80M.img7               76          90        7528+  83  Linux
./hd80M.img8               91         120       15088+  83  Linux
./hd80M.img9              121         162       21136+  83  Linux
```

▲图 13-9　fdisk 过程 4

```
Command (m for help): l   键入l查看已知文件系统id

 0  Empty           24  NEC DOS         81  Minix / old Lin bf  Solaris
 1  FAT12           39  Plan 9          82  Linux swap / So c1  DRDOS/sec (FAT-
 2  XENIX root      3c  PartitionMagic  83  Linux           c4  DRDOS/sec (FAT-
 3  XENIX usr       40  Venix 80286     84  OS/2 hidden C:  c6  DRDOS/sec (FAT-
 4  FAT16 <32M      41  PPC PReP Boot   85  Linux extended  c7  Syrinx
 5  Extended        42  SFS             86  NTFS volume set da  Non-FS data
 6  FAT16           4d  QNX4.x          87  NTFS volume set db  CP/M / CTOS / .
 7  HPFS/NTFS       4e  QNX4.x 2nd part 88  Linux plaintext de  Dell Utility
 8  AIX             4f  QNX4.x 3rd part 8e  Linux LVM       df  BootIt
 9  AIX bootable    50  OnTrack DM      93  Amoeba          e1  DOS access
 a  OS/2 Boot Manag 51  OnTrack DM6 Aux 94  Amoeba BBT      e3  DOS R/O
 b  W95 FAT32       52  CP/M            9f  BSD/OS          e4  SpeedStor
 c  W95 FAT32 (LBA) 53  OnTrack DM6 Aux a0  IBM Thinkpad hi eb  BeOS fs
 e  W95 FAT16 (LBA) 54  OnTrackDM6      a5  FreeBSD         ee  GPT
 f  W95 Ext'd (LBA) 55  EZ-Drive        a6  OpenBSD         ef  EFI (FAT-12/16/
10  OPUS            56  Golden Bow      a7  NeXTSTEP        f0  Linux/PA-RISC b
11  Hidden FAT12    5c  Priam Edisk     a8  Darwin UFS      f1  SpeedStor
12  Compaq diagnost 61  SpeedStor       a9  NetBSD          f4  SpeedStor
14  Hidden FAT16 <3 63  GNU HURD or Sys ab  Darwin boot     f2  DOS secondary
16  Hidden FAT16    64  Novell Netware  af  HFS / HFS+      fb  VMware VMFS
17  Hidden HPFS/NTF 65  Novell Netware  b7  BSDI fs         fc  VMware VMKCORE
18  AST SmartSleep  70  DiskSecure Mult b8  BSDI swap       fd  Linux raid auto
1b  Hidden W95 FAT3 75  PC/IX           bb  Boot Wizard hid fe  LANstep
1c  Hidden W95 FAT3 80  Old Minix       be  Solaris boot    ff  BBT
1e  Hidden W95 FAT1
```

▲图 13-10　fdisk 过程 5

```
Command (m for help): t   键入t设置分区id
Partition number (1-9): 5                                 把第5个分区id
Hex code (type L to list codes): 66                       (第1个逻辑分区)
Changed system type of partition 5 to 66 (Unknown)        设置为0x66

Command (m for help): t
Partition number (1-9): 6                                 把第6个分区id
Hex code (type L to list codes): 66                       (第2个逻辑分区)
Changed system type of partition 6 to 66 (Unknown)        设置为0x66

Command (m for help): t
Partition number (1-9): 7
Hex code (type L to list codes): 66
Changed system type of partition 7 to 66 (Unknown)

Command (m for help): t
Partition number (1-9): 8                                 ■
Hex code (type L to list codes): 66                       ■
Changed system type of partition 8 to 66 (Unknown)

Command (m for help): t                                   ■
Partition number (1-9): 9
Hex code (type L to list codes): 66
Changed system type of partition 9 to 66 (Unknown)

Command (m for help): p          键入p打印分区表，
                                 扩展分区中的全部逻辑分区id已经变成0x66
Disk ./hd80M.img: 0 MB, 0 bytes
16 heads, 63 sectors/track, 162 cylinders
Units = cylinders of 1008 * 512 = 516096 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0xfdf67f62

       Device Boot      Start         End      Blocks   Id  System
./hd80M.img1                1          32       16096+  83  Linux
./hd80M.img4               33         162       65520    5  Extended
./hd80M.img5               33          50        9040+  66  Unknown
./hd80M.img6               51          75       12568+  66  Unknown
./hd80M.img7               76          90        7528+  66  Unknown
./hd80M.img8               91         120       15088+  66  Unknown
./hd80M.img9              121         162       21136+  66  Unknown
```

▲图 13-11　fdisk 过程 6

```
Command (m for help): w            键入w将分区表写入硬盘并退出fdisk
The partition table has been altered!

Syncing disks.
[work@localhost bochs]$ fdisk -l hd80M.img      查看分区
You must set cylinders.
You can do this from the extra functions menu.

Disk hd80M.img: 0 MB, 0 bytes
16 heads, 63 sectors/track, 0 cylinders
Units = cylinders of 1008 * 512 = 516096 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0xfdf67f62

    Device Boot      Start         End      Blocks   Id  System
hd80M.img1               1          32       16096+  83  Linux
hd80M.img4              33         162       65520    5  Extended
hd80M.img5              33          50        9040+  66  Unknown
hd80M.img6              51          75       12568+  66  Unknown
hd80M.img7              76          90        7528+  66  Unknown
hd80M.img8              91         120       15088+  66  Unknown
hd80M.img9             121         162       21136+  66  Unknown
[work@localhost bochs]$ █
```

▲图 13-12　fdisk 过程 7

按照这样分区后，hd80M.img 的大小依然是 83607552 字节，原因是柱面数和磁头数是按照 bximage 给出的参数设置的，如果您感兴趣，可以在分区过程中把这两个参数改为其他较大值，分区结束后 hd80M.img 的尺寸将按照公式 3 变大。

好啦，本节到此结束，下节咱们再细说分区表。

13.1.3　磁盘分区表浅析

在上一节我们已经介绍了一部分有关分区的知识，还从硬件上解释了分区的本质，本节在软件上展开分区表的细节。

磁盘分区表（Disk Partition Table）简称 DPT，是由多个分区元信息汇成的表，表中每一个表项都对应一个分区，主要记录各分区的起始扇区地址，大小界限等。

分区表是由分区工具如 fdisk 创建的，但却是给操作系统使用的。听我这么一说，大伙儿不要误以为操作系统很“弱”，其实操作系统也可以创建分区表，它有底层硬件的一切操作权限，无所不能，只是在通常情况下操作系统直接安装在某个分区上，所以分区表要在内核安装之前建立好，因此分区工具通常独立于操作系统。有了分区表，操作系统（的文件系统）可以根据各表项中的信息对硬盘进行分区管理，只要按照表项中的信息访问磁盘，就不会出现分区越界的情况。分区表既然称为“表”，这表示各个表项的数据结构一致，因此磁盘分区表就是个数组，此数组长度固定为 4，数组元素是分区元信息的结构。

最初的磁盘分区表位于 MBR 引导扇区中，咱们先看看原汁原味的 MBR 引导扇区的逻辑结构。早在加载 loader 时就和大伙儿介绍过 MBR，MBR（Main Boot Record）即主引导记录，它是一段引导程序，其所在的扇区称为主引导扇区，该扇区位于 0 盘 0 道 1 扇区（物理扇区编号从 1 开始，逻辑扇区地址 LBA 从 0 开始），也就是硬盘最开始的扇区，扇区大小为 512 字节，这 512 字节内容由三部分组成。

（1）主引导记录 MBR。

（2）磁盘分区表 DPT。

（3）结束魔数 55AA，表示此扇区为主引导扇区，里面包含控制程序。

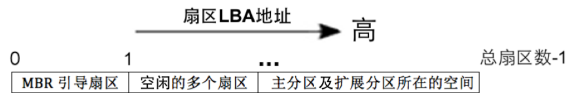
MBR 引导程序位于主引导扇区中偏移 0～0x1BD 的空间，共计 446 字节大小，这其中包括硬盘参数及部分指令（由 BIOS 跳入执行），它是由分区工具产生的，独立于任何操作系统。

磁盘分区表位于主引导扇区中偏移 0x1BE～0x1FD 的空间，总共 64 字节大小，每个分区表项是 16 字节，因此磁盘分区表最大支持 4 个分区。

魔数 55AA 作为主引导扇区的有效标志，位于扇区偏移 0x1FE～0x1FF，也就是最后 2 个字节。

以上这三部分便是 MBR 的主要结构。

在硬盘中，最开始的扇区是 MBR 引导扇区，接着是空闲的多个扇区，随后是具体的分区。它们的位置关系如图 13-13 所示。



▲图 13-13　硬盘布局

这个位于引导扇区后的“空闲的多个扇区”到底是做什么的？一般是多大？分区既然是由分区工具划分的，那这些空闲的扇区也必然是由分区工具有意空出的。前面在介绍分区的时候，和大伙儿介绍了“分区粒度”，也就是分区都要占用完整的柱面，柱面是由不同盘面上相同的磁道组成的，因此从定义上看，柱面肯定不能跨磁道（近而得出结论，同一个磁道也不能被多个分区共享）。而第 0 块又被 MBR 引导扇区占据了，因此 MBR 所在的磁道不能划入分区了，故分区起始地址要偏移磁盘 1 个磁道的大小，也就是一般为 63 扇区（在后面您就会看到分区起始是 0x3F，即 63）。总之，分区偏移 MBR 所在的磁道后，分区的起始地址便会是柱面的整数倍，这是由分区工具规划好的。小总结：对于不够一个柱面的剩余的空间一般不再利用，并不参与分区。除去 MBR 引导扇区占用的 1 扇区，这部分剩余空间是 62 个扇区。这个空间按理说不属于操作系统的范围，但操作系统确实是有能力支配它。由于仅仅是 62 个扇区的空间，能用它做的事情有限，因为空间太小会使扩展性很差，比如文件系统中的块位图大小是与分区大小成正比的，若将块位图存放在此处，受到这 62 个扇区的限制，能管理的分区范围也将大大缩水，因此很少有操作系统会用到这个磁道，我们也不用它。

最初的硬盘容量小，分区也不需要太多，整个硬盘使用 1 个分区表就够了，因此对最初的分区表来说，它的环境是整个硬盘，一个硬盘只有一个分区表，分区表所在的扇区被认为是整个硬盘最开始的扇区，这么说吧，站在分区表的角度，它会有这样的观念："整个硬盘就我这一个分区表，我所在的位置就是整个硬盘最开始的那个扇区"。注意，此"观念"是我杜撰的，为的是帮助解释子扩展分区，到后面您就会了解到我的良苦用心。

随着磁盘容量越来越大，对于磁盘管理来说，4 个分区显然不够用了，因此发明了扩展分区，寄望以此来解决原来所支持的分区数太少的问题。大伙儿知道，"分区表的长度固定"才是问题的症结所在。按理来说，扩展分区中包含多少个逻辑分区，扩展分区的分区表中就该有多少表项，可是任何时候新事物的发展都要把"向上兼容"当成头等大事，它既要兼容此固定长度为 4 个分区的分区表，又要突破固定分区数的限制，这似乎有点为难，该怎样设计扩展分区的分区表呢？一个两全其美的方案是视这个扩展分区为总扩展分区，将它划分成多个子扩展分区，每个子扩展分区"在逻辑上"相当于硬盘，因此每个子扩展分区都可以有 1 个分区表。这样一来，各个分区表的长度依然固定为 4，但是允许有无限多个分区表，分区表项多了，自然支持的分区数就多了。如何将这些分区表组织到一起呢？扩展分区表采用链式结构，将所有子扩展分区的分区表串在一起，形成可容纳无限个分区表的单向链表。链表是要有结点的，这里的每个分区表就是结点，一般的链表结点除了包括数据外，还必须要包括下一个结点的地址，分区表也采用了这种结构，其表项就分为两部分，一部分是描述逻辑分区的信息，另一部分是描述下一个子扩展分区的地址。

下面咱们顺着这个思路分析。扩展分区之下是子扩展分区，那这里所说的逻辑分区在哪里？

要想使用分区，就离不开分区表，逻辑分区也是分区，为了使用它，也需要有元信息来描述它的范围、边界、类型等信息，因此在子扩展分区中也要有分区表来描述这些逻辑分区。分区表本身也要在子扩展分区中占磁盘空间，因此实际情况是每个子扩展分区的空间并不是只有逻辑分区，在每个子扩展分区中最开始的扇区（剧透一下，此扇区称为 EBR 引导扇区，马上要介绍它）用于存储此子扩展分区中的分区表，此扇区中的内容也是前 446 字节是引导程序，中间 64 字节是分区表，后 2 字节是 0x55 和 0xAA，您看，它同 MBR 引导扇区的结构相同。紧随其后的是空闲的一部分扇区，其余剩下的大部分扇区才被用作存储数据的分区，即逻辑分区。因此，子扩展分区包含逻辑分区，一会咱们跟踪分区表您就会看得明明白白了。

为什么子扩展分区是这样的结构呢？现在要用到我之前杜撰的"观念"了。有了扩展分区后，出于兼容性考虑，这种"观念"依然被传承下来：扩展分区被划分出多个子扩展分区，每个子扩展分区都有自己的分区表，所以子扩展分区在逻辑上相当于单独的硬盘，各分区表在各个子扩展分区最开始的扇区中，该扇区同 MBR 引导扇区结构相同，由于是经扩展分区划分出来的，所以它们称为 EBR，即扩展引导记录。（现在想想看，主分区之所以称为"主"，也许是与其分区表项位于 MBR 引导扇区中有关吧。）MBR 只有 1 个，EBR 理论上有无限个，MBR 和 EBR 所在的扇区统称为引导扇区，它们的结构是相同的，MBR 中有的 EMR 中也有。我想这下您清楚了我为什么要把子扩展分区视为硬盘，因为各子扩展分区的结构也同整个硬盘结构一样，最开始的扇区都是引导扇区，中间都是空闲一小部分扇区，最后的大片扇区空间作为数据存储的分区。

现在大伙儿知道了，由于扩展分区采用了链式分区表，理论上可以存在无限个分区表，支持无限个逻辑分区。每一个逻辑分区所在的子扩展分区都有一个与 MBR 结构相同的 EBR，EBR 中分区表的第一分区表项用来描述所包含的逻辑分区的元信息，第二分区表项用来描述下一个子扩展分区的地址，第三、四表项未用到。位于 EBR 中的分区表相当于链表中的结点，第一个分区表项存的是分区数据，第二个分区表项存的是后继分区的指针。

值得一提的是这两个分区表项都是指向一个分区的起始，起始地址都是个扇区地址，只不过第一个分区表项指向的是该逻辑分区最开始的扇区，此扇区称为操作系统引导扇区，即 OBR 引导扇区。第二个分区表项指向下一个子扩展分区的 EBR 引导扇区。

这里可别搞混了，OBR 引导扇区不是 EBR 引导扇区。EBR 不属于分区之内，不属于操作系统管理的范围。而 OBR 引导扇区位于分区（主分区和逻辑分区）最开始的扇区，属于操作系统管理的范围。好在咱们在第 0 章中有介绍过 OBR、DBR、EBR 的内容，忘记的同学可以先看看。将来设计文件系统时会再碰到此扇区。

也许您觉得奇怪，同样结构的分区表项是如何存储这两种类型分区数据的，是时候介绍分区表项的结构了，请大伙儿见表 13-1。

表 13-1　　分区表项结构

偏　移　量	数 据 宽 度	描　　述
0	1	活动分区标记，此标记有两种取值，0x80 和 0。 0x80 表示活动分区，也就是此分区的引导扇区中包含引导程序，可引导。只要是可执行加载执行的程序都可作为引导程序，不要误以为引导程序一定得是内核提供的程序，尽管通常情况下都是内核加载器。 0 表示非活动分区，不可引导。 其他值非法
1	1	分区起始磁头号
2	1	分区起始扇区号
3	1	分区起始柱面号
4	1	文件系统类型 ID，如 0 表示不可识别的文件系统，1 表示 FAT32
5	1	分区结束磁头号
6	1	分区结束扇区号
7	1	分区结束柱面号
8	4	分区起始偏移扇区
12	4	分区容量扇区数

为了介绍清楚活动分区的意义，还得把有关 MBR、EBR 和 OBR 的内容再说说，不过有关它们更详细的内容还是以第 0 章的答疑为主。

活动分区是指引导程序所在的分区，活动分区标记是给 MBR 或其他需要移交 CPU 使用权的程序看的，它们通过此位来判断该分区的引导扇区中是否有可执行的程序，也就是引导程序，这个引导程序通常是操作系统内核加载器，故此引导程序通常被称为操作系统引导记录，即 OBR（OS Boot Record）。如果 MBR 发现该分区表项的活动分区标记为 0x80，这就表示该分区的引导扇区中有引导程序（这是 MBR 与分区工具或操作系统约定好的），MBR 就将 CPU 使用权交给此引导程序，如果此引导程序是操作系统或其加载器，此时操作系统便掌握了 CPU 使用权，也就是大家平时所说的加载内核。这里一直说的“分区引导扇区”是位于分区最开始的扇区，是分区引导程序所在的扇区，由于此引导程序通常都是操作系统内核加载器，故此扇区被称为操作系统引导扇区，也就是 OBR 所在的扇区，即 OBR 引导扇区。注意啦，OBR 引导扇区并不是 EBR 或 MBR 引导扇区，它们虽然都包含引导程序，并且都以 0x55 和 0xaa 结束，但它们最大的区别是分区表只包含在 MBR 和 EBR 中，OBR 中可没有分区表。MBR 和 EBR 所在的扇区不属于分区范围之内，它们是由分区工具创建并管理的，因此不归操作系统管理，操作系统不可以随意往里面写数据，尽管操作系统有能力这样做。而 OBR 引导扇区是分区中最开始的扇区，归操作系统的文件系统管理，因此操作系统通常往 OBR 引导扇区中添加内核加载器的代码，供 MBR 调用以实现操作系统自举，总之 OBR 引导扇区中绝对不包括分区表。

文件系统类型是指 NTFS、FAT32、EXT2 等，我们在 fdisk 过程中用 l 命令列出的便是。

这里咱们重点关注“分区起始偏移扇区”和“分区容量扇区数”。

“分区起始偏移扇区”是个相对量，它表示各分区的起始扇区地址是相对于某“基准”的偏移扇区数，各分区的绝对扇区 LBA 地址=“基准”的绝对扇区起始 LBA 地址+各分区的起始偏移扇区，这个“基准”是指分区所依赖的上层对象，或者说是创建该分区的父对象。我知道这么还是说太抽象了，有必要再深入讨论。

先梳理下分区层次关系，前面已述，总扩展分区被直接拆分成多个子扩展分区，子扩展分区又被拆分成 EBR 引导扇区、空闲扇区和逻辑分区三部分。“基准”也因分区类型而异。

对于逻辑分区而言，逻辑分区是在子扩展分区中拆分出来的，其具体地址依赖于子扩展分区自身的起始地址，因此逻辑分区的基准是子扩展分区的起始扇区 LBA 地址。

对于子扩展分区而言，子扩展分区是在总扩展分区中拆分出来的，其具体地址依赖于总扩展分区自身

的起始地址，因此子扩展分区的基准是总扩展分区的起始扇区 LBA 地址。

对于主分区或总扩展分区而言，这两类分区本身是独立、无依赖的分区，因此基准为 0。此概念咱们在实践中去理解。

“分区容量扇区数”意义比较明确，就是表示分区的容量扇区数。

以上两项用来确定一个分区的位置和大小。

以上介绍的内容不多，不知您是不是感觉有点晕？觉得有些概念如逻辑分区、扩展分区可能和您平时理解的不一样？说实话，如果您觉得费解的话，我还是觉得有些欣慰的，如果这些概念真像我们平时所想象的那样，我还真不值得花这份心思写这一节。虽然分区表不难理解，但把它的结构说清楚还真的不容易，主要是小弟我实在是才疏学浅，有一些内容我没有找到对应的术语，不知该怎样去表达它们，甚至我自创了一些“观念”来帮助表达，仅这一小节我都用了 3 天时间还没写完。个人觉得，如果不亲自跟一下分区表的话还是会觉得云里雾里，还是在实践中理解吧，咱们跟一下在上节中创建的分区表，通过逐字节地分析，我想大伙儿一定会弄清楚的。要以字节方式查看文件内容，咱们要借助 xxd.sh 脚本了，大家对它应该很熟悉了，它只是 xxd 命令的封装，脚本用法是：

xxd.sh 待查看的文件起始字节偏移查看的字节数

先查看主引导扇区，也就是硬盘最开始的扇区，如图 13-14 所示。

```
[work@localhost bochs]$ sh ~/tool/xxd.sh hd80M.img 0 512
0000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00001b0: 00 00 00 00 00 00 00 00 62 7F F6 FD 00 00 00 01  ........b.......
00001c0: 01 00 83 0F 3F 1F 3F 00 00 00 C1 7D 00 00 00 00  ....?.?....}....
00001d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00001e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00001f0: 01 20 05 0F 3F A1 00 7E 00 00 E0 FF 01 00 55 AA  . ..?..~......U.
[work@localhost bochs]$
```

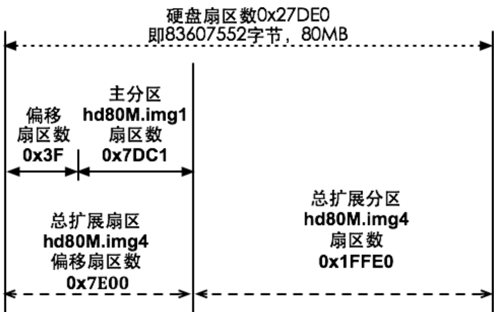
▲图 13-14　主引导扇区

您看，地址 0～0x1b0 之间的数字全是 0，xxd 已经在输出中将其省略了，在偏移 0x1be 的地方才是分区表，一直到 0x1fd，一共是 64 字节，最后才是魔数 55 AA。在分区表中只创建了 2 个分区，还记得吗？第 1 分区咱们用来创建主分区，第 4 分区用来创建扩展分区，第 2～3 分区咱们没占用，因此是一系列的 0 值。第 1 分区和第 4 分区的元信息已经按照分区表项结构给大伙儿标出来了，咱们关注的重点信息是分区类型，分区起始偏移扇区（简称偏移扇区）和分区容量扇区数（简称扇区数），将它们汇总在表 13-2 中。

表 13-2　　主引导扇区中分区表部分元信息

分　区	分区类型	偏移扇区	扇区数
主分区 hd80M.img1	0x83	0x0000003F	0x00007DC1
总扩展分区 hd80M.img4	0x05	0x00007E00	0x0001FFE0

第 1 分区是主分区 hd80M.img1，其类型是 0x83，属于 Linux 文件系统类型，偏移扇区是 0x3f，扇区数是 0x7dc1。第 4 分区是总扩展分区 hd80M.img4，其类型是 0x05，这说明它确实是扩展分区，偏移扇区是 0x7e00，扇区数是 0x1ffe0。这两个分区在硬盘中的布局如图 13-15 所示。



▲图 13-15　分区布局 1

图 13-15 中的分区属于同一个分区表，位于横向实线箭头上的是主分区，箭头下的是扩展分区。约定下，在后面的分区布局图中，我们把隶属于同一个分区表的分区用横向虚线箭头框出来，只要在两个横向虚线间的分区就是同一个分区表。

这里涉及到了偏移扇区的“基准”。主分区和总扩展分区的起始扇区是以 0 为基准的，比如主分区偏移扇区是 0x3f，该分区的绝对扇区 LBA 地址也是 0x3f，扩展分区的偏移扇区 0x7e00 也是如此。以上是特指主分区和扩展分区的情况，到了逻辑分区就不一样了，碰到时再说。

MBR 引导扇区中的分区表查看完了，下面查看扩展分区的分区表。扩展分区中的所有分区表被组织成单向链表，咱们查看链表中的第 1 个结点，也就是第 1 个子扩展分区的 EBR 引导扇区，为方便起见，暂且

称之为 EBR1。总扩展分区的起始扇区地址是 0x7e00，将其乘以 512 换算为字节，如图 13-16 所示。

```
[work@localhost ~]$ sh ~/tool/calculator.sh '0x00007E00*512' x
fc0000
[work@localhost ~]$
[work@localhost bochs]$ sh ~/tool/xxd.sh hd80M.img 0xfc0000 512
0fc0000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
0fc01b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01  ................
0fc01c0: 01 20 66 0F 3F 31 3F 00 00 00 A1 46 00 00 00 00  . f.?1?....F....
0fc01d0: 01 32 05 0F 3F 4A E0 46 00 00 70 62 00 00 00 00  .2..?J.F..pb....
0fc01e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0fc01f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA  ..............U.
[work@localhost bochs]$
```

▲图 13-16　EBR1

该分区表中有两个分区表项，第 1 个表项在此扇区中偏移范围是 0x1be～0x1cd，它是逻辑分区 hd80M.img5 的元信息，第 2 个表项在此扇区中偏移范围是 0x1ce～0x1dd，这是下一个扩展分区的扇区 LBA 地址。这下实实在在地让您看到了，子扩展分区中的分区表结构是典型的单向链表结点。将部分信息汇总到表 13-3 中。

表 13-3　　第 1 个子扩展分区表部分信息

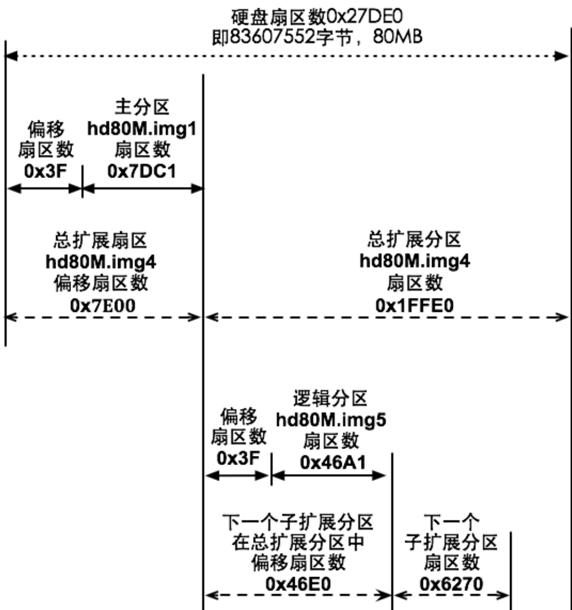
分 区 编 号	分 区 类 型	偏 移 扇 区	扇　区　数
逻辑分区 hd80M.img5	0x66	0x0000003F	0x000046A1
下一个子扩展分区	0x05	0x000046E0	0x00006270

子扩展分区是在总扩展分区中创建的，子扩展分区的偏移扇区理应以总扩展分区的绝对扇区 LBA 地址为基准，因此，“子扩展分区的绝对扇区 LBA 地址=总扩展分区绝对扇区 LBA 地址+子扩展分区的偏移扇区”。

逻辑分区是在子扩展分区中创建的，逻辑分区的偏移扇区理应以子扩展分区的绝对扇区 LBA 地址为基准，因此，“逻辑分区的绝对扇区 LBA 地址=子扩展分区绝对扇区 LBA 地址+逻辑分区偏移扇区”。这里的子扩展分区就是当前子扩展分区。

表 13-3 中，逻辑分区 hd80M.img5 的分区类型是 0x66，这是咱们在 fdisk 中设置的，其偏移是 0x3f，这是指在当前子扩展分区中的偏移扇区数，当前子扩展分区地址是由前驱结点（分区表）中的“下一个子扩展分区的位移扇区“+”总扩展分区绝对扇区 LBA 地址”，但由于前驱结点是总扩展分区，因此 hd80M.img5 的绝对扇区地址是 0x7e00+0x3f=0x7e3f。

到目前为止，分区布局如图 13-17 所示。



▲图 13-17　分区布局 2

如前约定，两个横向虚线箭头间的分区属于同一个分区表，图 13-17 中上面那对虚线间的分区是上次介绍过的分区，咱们以增量方式绘制分区布局，每次新增的分区放在最下面。从这张图中可以清晰看出，逻辑分区是在上一个分区表中的子扩展分区中。

当前结点中的“下一个子扩展分区”是下一个逻辑分区所在的子扩展分区，其偏移扇区是 0x000046E0，因此其绝对扇区地址要加上总扩展分区扇区地址，即 0x7e00+0x46E0= 0xC4E0。将其转换成字节后继续查看该扇区，如图 13-18 所示。

```
[work@localhost ~]$ sh ~/tool/calculator.sh '(0x00007E00+0x000046E0)*512' x
189c000
[work@localhost ~]$
[work@localhost bochs]$ sh ~/tool/xxd.sh hd80M.img 0x189c000 512
189c000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
189c1b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01  ................
189c1c0: 01 32 66 0F 3F 4A 3F 00 00 00 31 62 00 00 00 00  .2f.?J?...1b....
189c1d0: 01 4B 05 0F 3F 59 50 A9 00 00 10 3B 00 00 00 00  .K..?YP....;....
189c1e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
189c1f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA  ..............U.
[work@localhost bochs]$
```

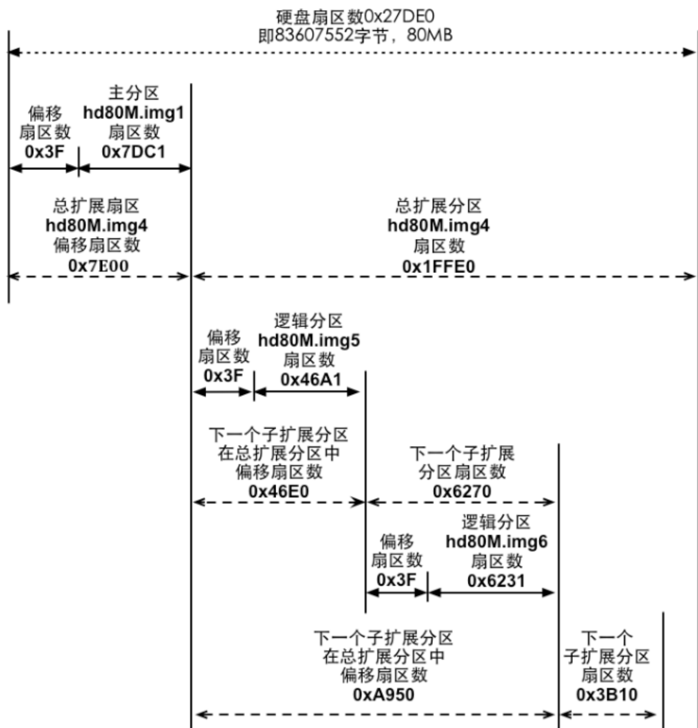
▲图 13-18　EBR2

同前面一样，第 1 个表项是逻辑分区，第 2 个表项是扩展分区，重点信息汇总到表 13-4 中。

表 13-4　　　　　　　　　　　　　　　第 2 个子扩展分区表部分信息

分 区 编 号	分 区 类 型	偏 移 扇 区	扇　区　数
逻辑分区 hd80M.img6	0x66	0x0000003F	0x00006231
下一个子扩展分区	0x05	0x0000A950	0x00003B10

最新分区布局如图 13-19 所示。



▲图 13-19　分区布局 3

表 13-4 中下一个子扩展分区偏移扇区是 0x0000A950，为了定位该分区，依然重复之前的过程，将其加上总扩展分区地址 0x7e00，然后转换为字节，EBR3 内容如图 13-20 所示。

```
[work@localhost ~]$ sh ~/tool/calculator.sh '(0x00007E00+0x0000A950)*512' x
24ea000
[work@localhost ~]$
[work@localhost bochs]$ sh ~/tool/xxd.sh hd80M.img 0x24ea000 512
24ea000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
24ea1b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01  ................
24ea1c0: 01 4B 66 0F 3F 59 3F 00 00 00 D1 3A 00 00 00 00  .Kf.?Y?....:....
24ea1d0: 01 5A 05 0F 3F 77 60 E4 00 00 20 76 00 00 00 00  .Z..?w`... v....
24ea1e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
24ea1f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA  ..............U.
[work@localhost bochs]$
```

▲图 13-20　EBR3

提取重点信息到表 13-5 中。

表 13-5　　　　　　　　　　　　　　　第 3 个子扩展分区表部分信息

分 区 编 号	分 区 类 型	偏 移 扇 区	扇　区　数
逻辑分区 hd80M.img7	0x66	0x0000003F	0x00003AD1
下一个子扩展分区	0x05	0x0000E460	0x00007620

分区布局图不再单独列出了，等剩下的几个分区表都查看完再列个分区布局汇总图。图 13-21 所示是 EBR4 引导扇区，计算过程您懂的。重点信息见表 13-6。

表 13-6　　第 4 个子扩展分区表部分信息

分 区 编 号	分 区 类 型	偏 移 扇 区	扇　区　数
逻辑分区 hd80M.img8	0x66	0x0000003F	0x000075E1
下一个子扩展分区	0x05	0x00015A80	0x0000A560

图 13-22 所示是 EBR5 所在扇区。

```
[work@localhost ~]$ sh ~/tool/calculator.sh '(0x00007E00+0x0000E460)*512' x
2c4c000
[work@localhost ~]$
[work@localhost bochs]$ sh ~/tool/xxd.sh hd80M.img 0x2c4c000 512
2c4c000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
2c4c1b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01  ................
2c4c1c0: 01 5A 66 0F 3F 77 3F 00 00 00 E1 75 00 00 00 00  .Zf.?w?....u....
2c4c1d0: 01 78 05 0F 3F A1 80 5A 01 00 60 A5 00 00 00 00  .x..?..Z..`.....
2c4c1e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
2c4c1f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA  ..............U.
[work@localhost bochs]$
```

▲图 13-21　EBR4

```
[work@localhost ~]$ sh ~/tool/calculator.sh '(0x00007E00+0x00015A80)*512' x
3b10000
[work@localhost ~]$
[work@localhost bochs]$ sh ~/tool/xxd.sh hd80M.img 0x3b10000 512
3b10000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
3b101b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01  ................
3b101c0: 01 78 66 0F 3F A1 3F 00 00 00 21 A5 00 00 00 00  .xf.?.?...!.....
3b101d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
3b101e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
3b101f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA  ..............U.
[work@localhost bochs]$
```

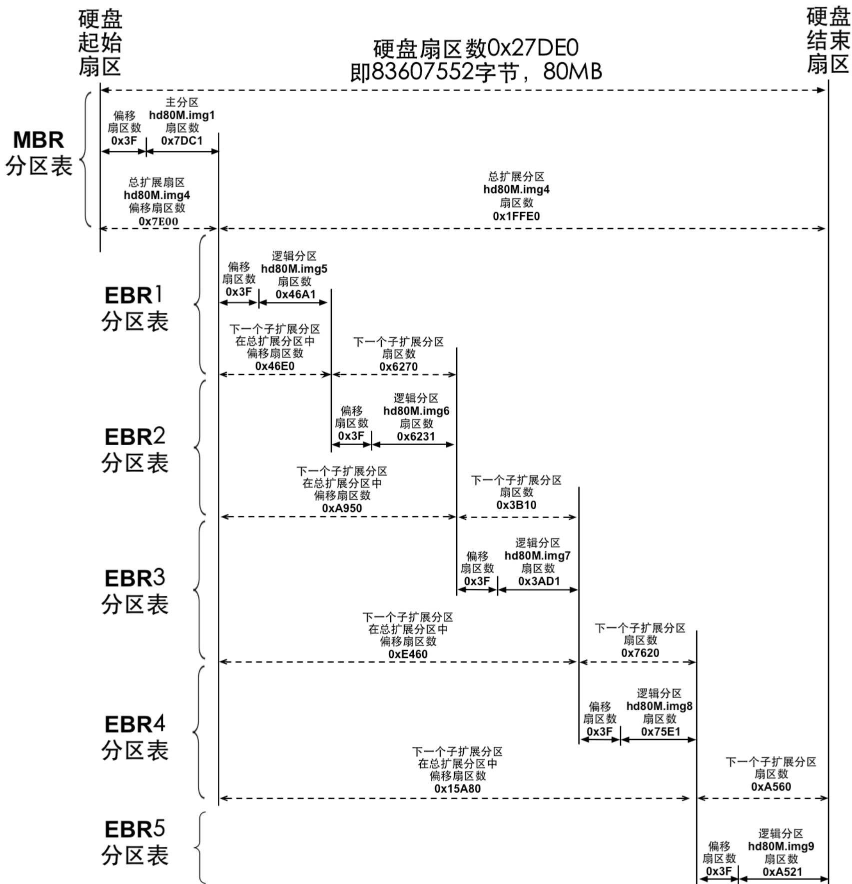
▲图 13-22　EBR5

EBR5 中只有一个逻辑分区，因此分区结束。重点信息见表 13-7。

表 13-7　　第 5 个子扩展分区表部分信息

分 区 编 号	分 区 类 型	偏 移 扇 区	扇　区　数
逻辑分区 hd80M.img9	0x66	0x0000003F	0x0000A521

图 13-23 是硬盘 hd80M.img 所有分区布局图。



▲图 13-23　分区布局汇总

本节终于结束了，兄弟们也辛苦了，有关分区的理论知识也就到此为止，下节咱们开始写硬盘驱动。

13.2 编写硬盘驱动程序

什么是驱动程序？问了下度娘，上面说驱动程序是让计算机和硬件通信的特殊程序，是“硬件的灵魂”、“操作系统和硬件之间的桥梁”。这话对是对，但对于初学者来说总感觉和没说一样，还是没懂驱动程序是什么。

大伙儿知道，硬件是独立的个体，它提供一套方法作为操作接口给外界调用，但此接口往往是最原始、最简陋、最繁琐的，相对咱们习惯的高级语言来说，这些接口使用起来非常麻烦，很多指令要提前设置好各种参数，基本上都是要用汇编语言来操作寄存器。基于此描述，对于驱动程序我个人的理解是驱动程序是对硬件接口的封装，它把参数设置等重复、枯燥、复杂的过程封装成一个过程，避免每次执行命令时都重复做这些工作，根据需要也可以提供相关的策略，如缓存等，让硬件操作更加容易、省事、方便，无需再显式做一些底层设置。因此没有驱动程序的话，操作系统也是可以同硬件交流的，无非是直接操作 IO 端口。举个例子，平时咱们自己做饭吃，要经过买菜、洗菜、切菜、炒菜的过程，有人嫌这太麻烦了（吃饭用 5 分钟要做 1 小时），干脆就去饭店吃吧，直接吃就行了，免去了买菜、砍价、配菜、烹饪的过程，饭店起的作用就类似于驱动程序。

13.2.1 硬盘初始化

硬件是实实在在的东西，要想在软件中管理它们，只能从逻辑上抓住这些硬件的特性，将它们抽象成一些数据结构，然后这些数据结构便代表了硬件，用这些数据结构来组织硬件的信息及状态，在逻辑上硬件就是这数据结构。硬盘也是“实在”的东东，为了管理它们还是得将它们抽象成某些数据结构，这就是本节的任务之一。

为了支持硬盘操作，咱们还有几件事要做。硬盘上有两个 ata 通道，也称为 IDE 通道。第 1 个 ata 通道上的两个硬盘（主和从）的中断信号挂在 8259A 从片的 IRQ14 上，没错，是两个硬盘共享同一个 IRQ 接口，也许您在想，硬件发生中断时，如何区分中断是来自哪个硬盘的？是这样的，硬盘发生中断的条件是咱们对硬件执行了某些命令，然后硬盘完成任务后才发中断，咱们在对硬盘发命令时，需要提前指定是对主盘，还是从盘操作，这是在硬盘控制器的 device 寄存器中第 4 位的 dev 位指定的（忘记的话回头看看第 3 章的硬盘控制器端口），因此自然就知道是哪个硬盘来了中断信号，具体的作法咱们在后面实现的部分再说。顺便说一下，第 2 个 ata 通道接在 8259A 从片的 IRQ15 上，该 ata 通道上可支持两个硬盘。来自 8259A 从片的中断是由 8259A 主片帮忙向处理器传达的，8259A 从片是级联在 8259A 主片的 IRQ2 接口的，因此为了让处理器也响应来自 8259A 从片的中断，屏蔽中断寄存器必须也把 IRQ2 打开，如代码 13-1 所示。

代码 13-1　（project/c13/a/kernel/interrupt.c）

```
…略
41 /* 初始化可编程中断控制器 8259A */
42 static void pic_init(void) {
…略
56 /* IRQ2 用于级联从片，必须打开，否则无法响应从片上的中断。
57  主片上打开的中断有 IRQ0 的时钟，IRQ1 的键盘和级联从片的 IRQ2，
    其他全部关闭 */
58    outb (PIC_M_DATA, 0xf8);
59
60 /* 打开从片上的 IRQ14，此引脚接收硬盘控制器的中断 */
61    outb (PIC_S_DATA, 0xbf);
62
63    put_str("   pic_init done\n");
64 }
```

代码第 58～61 行重新设置了中断屏蔽寄存器，目前是在主片 8259A 上打开的中断 IRQ0 的时钟、IRQ1 的键盘和级联从片的 IRQ2。从片 8259A 上打开的中断是 IRQ14 的硬盘。

顺便提一句，在中断处理程序中，如果中断源是来自从片 8259A 的话，在发送中断结束信号 EOI 的时候，主片和从片都要发送。否则，将无法继续响应新的中断。不过咱们的中断处理程序一直都是向主从两片 8259A 同时发送 EOI。

还有一件小事要做，完全是为了让咱们自己方便。目前咱们在内核中打印输出时，都是用 console_put_xxx 之类的函数，可是自从咱们为用户进程实现了 printf 函数后，越来越觉得 console_put_xxx 系列的函数不方便，为此咱们也心疼下自己，在内核中实现格式化输出函数 printk。printk 和 printf 原理是一样的，只是 printk 是专门用在内核态的格式化输出函数，它是 printf 的孪生兄弟。由于是在内核态下实现，因此不需要系统调用啦，所以实现起来更简单了，我们把它定义在 lib/kernel/stdio.kernel.c 中，如代码 13-2 所示。

代码 13-2　（project/c13/a/lib/kernel/stdio-kernel.c）

```
…略
10 /* 供内核使用的格式化输出函数 */
11 void printk(const char* format, ...) {
12     va_list args;
13     va_start(args, format);
14     char buf[1024] = {0};
15     vsprintf(buf, format, args);
16     va_end(args);
17     console_put_str(buf);
18 }
```

下面介绍和硬盘相关的数据结构了，它定义在 device/ide.h 中，这是我们新创建的文件，如代码 13-3 所示。

代码 13-3　（project/c13/a/device/ide.h）

```
…略
 7 /* 分区结构 */
 8 struct partition {
 9     uint32_t start_lba;            // 起始扇区
10     uint32_t sec_cnt;              // 扇区数
11     struct disk* my_disk;          // 分区所属的硬盘
12     struct list_elem part_tag;     // 用于队列中的标记
13     char name[8];                  // 分区名称
14     struct super_block* sb;        // 本分区的超级块
15     struct bitmap block_bitmap;    // 块位图
16     struct bitmap inode_bitmap;    // i 结点位图
17     struct list open_inodes;       // 本分区打开的 i 结点队列
18 };
19
20        /* 硬盘结构 */
21        struct disk {
22        char name[8];          // 本硬盘的名称
23        struct ide_channel* my_channel;       // 此块硬盘归属于哪个 ide 通道
24        uint8_t dev_no;                       // 本硬盘是主 0，还是从 1
25        struct partition prim_parts[4];       // 主分区顶多是 4 个
26        struct partition logic_parts[8];
          // 逻辑分区数量无限，但总得有个支持的上限，那就支持 8 个
27 };
28
29        /* ata 通道结构 */
30        struct ide_channel {
31        char name[8];               // 本 ata 通道名称
32        uint16_t port_base;         // 本通道的起始端口号
33        uint8_t irq_no;             // 本通道所用的中断号
34        struct lock lock;           // 通道锁
35        bool expecting_intr;        //表示等待硬盘的中断
36        struct semaphore disk_done;  // 用于阻塞、唤醒驱动程序
37        struct disk devices[2];      // 一个通道上连接两个硬盘，一主一从
38 };
```

程序开头定义的 struct partition 是分区表结构，成员 start_lba 表示分区的起始扇区，sec_cnt 是分区的容量扇区数，一个硬盘有很多分区，因此成员 my_disk 表示此扇区属于哪个硬盘。part_tag 是本分区的标记，将来会将分区汇总到队列中，需要用此标记。name 是分区名称，如 sda1、sda2。第 14 行的“struct super_block* sb”，它是超级块指针，有关这部分咱们在后面介绍，此处只是用来占个位置，当前节的代码中并没有定义超级块的结构，因此更不可能包含超级块的头文件，因为这里的超级块是 struct super_block*指针，32 位系统下指针都是 32 位，在有指针操作的时候才会涉及到数据宽度，而本节咱们并没有使用它，因此编译并没有问题。后面的 3 个成员是文件系统中涉及的。为了减少对低速磁盘的访问次数，文件系统通常以

多个扇区为单位来读写磁盘，这多个扇区就是块。block_bitmap 是块位图，用来管理本分区所有块，为了简单，咱们的块大小是由 1 个扇区组成的，所以 block_bitmap 就是管理扇区的位图。inode_bitmap 是 i 结点管理位图。open_inodes 是分区所打开的 inode 队列。这些内容要在文件系统的部分中介绍，现在多说无益，此处知道在分区中有超级块指针即可。

在下面定义的是 struct disk，此结构体代表硬盘，name 表示硬盘的名称，比如 sda、sdb 等。一个通道上有两块硬盘，所以 my_channel 用于指定本硬盘所属的通道，成员 dev_no 用来表示本硬盘是主盘，还是从盘，prim_parts 是本硬盘中的主分区数量，最多是 4 个主分区，logic_parts 是逻辑分区的数量，咱们这里限制为 8 个。

接下来定义的是 struct ide_channel，此结构表示 ide 通道，也就是 ata 通道。

成员 name 是通道的名称，如 ata0 或 ide0。

port_base 是本通道的端口基址，对于它要多解释两句，咱们这里只处理两个通道的主板，每个通道的端口范围是不一样的，通道 1（Primary 通道）的命令块寄存器端口范围是 0x1F0～0x1F7，控制块寄存器端口是 0x3F6，通道 2（Secondary 通道）命令块寄存器端口范围是 0x170～0x177，控制块寄存器端口是 0x376。通道 1 的端口可以以 0x1F0 为基数，其命令块寄存器端口在此基数上分别加上 0～7 就可以了，控制块寄存器端口在此基数上加上 0x206，同理，通道 2 的基数就是 0x170。

成员 irq_no 是本通道的中断号，在硬盘的中断处理程序中要根据中断号来判断在哪个通道中操作，将来实现硬盘中断处理程序时就清楚了。

成员 lock 是本通道的锁，用来实现通道的互斥。为什么要给通道加个锁呢？大伙儿知道，1 个通道中有主、从两块硬盘，向硬盘下达命令的时候可以通过 device 寄存器中的 dev 位来指定操作哪块硬盘，这很好区分。但硬盘完成操作后，它还得通知调用者任务执行的结果，是顺利完成了，还是失败了，如果是读硬盘的话，现在可以取数据了，这里是让硬盘主动发中断来通知调用者的。可是一个通道只能有 1 个中断信号，因此通道中的两个硬盘也只能共用同一个中断，中断发生时，中断处理程序是如何区分中断信号来自哪一个硬盘呢？还真不知道怎样区分，所以一次只允许通道中的 1 个硬盘操作，因此在通道中设置锁来实现互斥，对通道中任何一个硬盘操作时都要申请该锁以实现独享通道。

成员 expecting_intr 表示本通道正等待硬盘中断。驱动程序向硬盘发完命令后等待来自硬盘的中断，中断处理程序中会通过此成员来判断此次的中断是否是因为之前的硬盘操作命令引起的，如果是，则进行下一步动作，如获取数据等。

成员 disk_done 是个信号量，它的作用是：驱动程序向硬盘发送命令后，在等待硬盘工作期间可以通过此信号量阻塞自己，避免干等着浪费 CPU。等硬盘工作完成后会发出中断，中断处理程序通过此信号量将硬盘驱动程序唤醒。

成员 devices 是个长度为 2 的数组，表示一个通道中的两个硬盘。

头文件就这样，下面介绍 ide.c 中的实现，请见代码 13-4。

代码 13-4　（project/c13/a/device/ide.c）

```
…略
10 /* 定义硬盘各寄存器的端口号 */
11 #define reg_data(channel)       (channel->port_base + 0)
12 #define reg_error(channel)      (channel->port_base + 1)
13 #define reg_sect_cnt(channel)  (channel->port_base + 2)
14 #define reg_lba_l(channel)     (channel->port_base + 3)
15 #define reg_lba_m(channel)      (channel->port_base + 4)
16 #define reg_lba_h(channel)     (channel->port_base + 5)
17 #define reg_dev(channel)        (channel->port_base + 6)
18 #define reg_status(channel)    (channel->port_base + 7)
19 #define reg_cmd(channel)         (reg_status(channel))
20 #define reg_alt_status(channel) (channel->port_base + 0x206)
21 #define reg_ctl(channel)         reg_alt_status(channel)
22
23 /* reg_alt_status 寄存器的一些关键位 */
24 #define BIT_ALT_STAT_BSY       0x80    // 硬盘忙
25 #define BIT_ALT_STAT_DRDY      0x40    // 驱动器准备好
26 #define BIT_ALT_STAT_DRQ       0x8     // 数据传输准备好了
27
```

```
28 /* device 寄存器的一些关键位 */
29 #define BIT_DEV_MBS    0xa0      // 第 7 位和第 5 位固定为 1
30 #define BIT_DEV_LBA    0x40
31 #define BIT_DEV_DEV    0x10
32
33 /* 一些硬盘操作的指令 */
34 #define CMD_IDENTIFY      0xec    // identify 指令
35 #define CMD_READ_SECTOR    0x20    // 读扇区指令
36 #define CMD_WRITE_SECTOR   0x30    // 写扇区指令
37
38 /* 定义可读写的最大扇区数，调试用的 */
39 #define max_lba ((80*1024*1024/512) - 1)  // 只支持 80MB 硬盘
40
41 uint8_t channel_cnt;      // 按硬盘数计算的通道数
42 struct ide_channel channels[2];  // 有两个 ide 通道
43
44 /* 硬盘数据结构初始化 */
45 void ide_init() {
46     printk("ide_init start\n");
47     uint8_t hd_cnt = *((uint8_t*)(0x475));     // 获取硬盘的数量
48     ASSERT(hd_cnt > 0);
49     channel_cnt = DIV_ROUND_UP(hd_cnt, 2);
          // 一个 ide 通道上有两个硬盘，根据硬盘数量反推有几个 ide 通道

50     struct ide_channel* channel;
51     uint8_t channel_no = 0;
52
53     /* 处理每个通道上的硬盘 */
54     while (channel_no < channel_cnt) {
55         channel = &channels[channel_no];
56         sprintf(channel->name, "ide%d", channel_no);
57
58         /* 为每个 ide 通道初始化端口基址及中断向量 */
59         switch (channel_no) {
60             case 0:
61                channel->port_base   = 0x1f0;
                              // ide0 通道的起始端口号是 0x1f0

62                channel->irq_no     = 0x20 + 14;
         // 从片 8259a 上倒数第二的中断引脚
                       // 硬盘，也就是 ide0 通道的中断向量号
63                break;
64             case 1:
65                channel->port_base   = 0x170;
             // ide1 通道的起始端口号是 0x170

66                channel->irq_no     = 0x20 + 15;
       // 从 8259A 上的最后一个中断引脚
                              // 我们用来响应 ide1 通道上的硬盘中断
67                break;
68         }
69
70         channel->expecting_intr = false;
         // 未向硬盘写入指令时不期待硬盘的中断
71        lock_init(&channel->lock);
72
73      /* 初始化为 0，目的是向硬盘控制器请求数据后，
        硬盘驱动 sema_down 此信号量会阻塞线程，
74      直到硬盘完成后通过发中断，
        由中断处理程序将此信号量 sema_up，唤醒线程 */
75         sema_init(&channel->disk_done, 0);
76         channel_no++;        // 下一个 channel
77      }
78      printk("ide_init done\n");
79 }
```

在代码第 11～21 行定义的宏是针对两个 ata 通道不同寄存器的端口，在第 3 章已经介绍了硬盘的端口，大伙儿可以再参考下“表 3-17 硬盘控制器主要端口寄存器”。

接下来定义了寄存器中的关键位，命名规则是 BIT_寄存器名称_位名。第 23～26 行定义的是 status 寄存器中的一些关键位，可以对照“图 3-32 status 寄存器”了解下，第 28～31 行定义的是 device 寄存器

中的一些关键位，可以对照“图 3-31 device 寄存器”了解下，顺便说一句，BIT_DEV_MBS 是指 device 中的第 5 位和第 7 位，这两位固定为 1。

下面是一些硬盘操作的命令，这里就定义了三个，对咱们的简单应用来说已经够用了。分别是 identify 命令，其值为 0xec，读扇区命令，其值为 0x20，写扇区指令，其值为 0x30。其中 identify 指令用来获取硬盘的身份信息，下节我们就要用到它。

下面定义的宏 max_lba 表示最大的 lba 地址，这是调试用的，避免出现扇区地址计算错误而出现越界的情况。

变量 channel_cnt 用来表示机器上的 ata 通道数，这里咱们的做法比较“鲁莽”，仅仅是通过硬盘数来反推有几个通道，也就是硬盘数除以 2 便是通道数。下面的 channels 是通道数组。

最后是硬盘初始化函数 ide_init，此函数目前所做的工作是初始化以上定义的数据结构，将来还要注册硬盘中断处理程序、检测硬盘参数和扫描分区表。

第 47 行通过代码“*((uint8_t*)(0x475))”在地址 0x475 处获取硬盘的数量，将其存入变量 hd_cnt 中。低端 1MB 以内的虚拟地址和物理地址相同，所以虚拟地址 0x475 可以正确访问到物理地址 0x475。

第 49 行通过代码“DIV_ROUND_UP(hd_cnt, 2);”简单地推算出了通道数，存入变量 channel_cnt 中。

第 54 行通过 while 循环处理每一个通道，在 switch 结构中针对这两个通道依次初始化端口基址 port_base 和中断号 irq_no。switch 结构的好处是便于扩展，虽然个人 PC 一般都是两个 ide 通道的主板，但多个 ide 通道的服务器确实很普遍，当有新的通道加入时，再加个 case 分支就可以了。这里通道 1 端口基址是 0x1f0，中断号是 0x2e，即 0x20 + 14。通道 2 的端口基址是 0x170，中断号是 0x2f，即 0x20 + 15。

下面初始化通道 expecting_intr 为 false，再初始化通道的锁 channel->lock 和信号量 channel-> disk_done。

数据结构至此初始化完成，本节偏重于基础构建，没啥实际功能验证，因此，本节的内容就到这了，大伙儿辛苦了。

13.2.2　实现 thread_yield 和 idle 线程

在进行下一步之前，我们先要完善一些基础构件，本节完成 thread_yield 和 idle 线程。

thread_yield 定义在 thread.c 中，它的功能是主动把 CPU 使用权让出来，它与 thread_block 的区别是 thread_yield 执行后任务的状态是 TASK_READY，即让出 CPU 后，它会被加入到就绪队列中，下次还能继续被调度器调度执行，而 thread_block 执行后任务的状态是 TASK_BLOCKED，需要被唤醒后才能加入到就绪队列，所以下次执行还不知道是什么时候。

好啦，下面看 thread_yield 的实现，见代码 13-5。

代码 13-5　（project/c13/b/thread/thread.c）

```
…略
 13 struct task_struct* idle_thread;    // idle 线程
…略
 21 /* 系统空闲时运行的线程 */
 22 static void idle(void* arg UNUSED) {
 23     while(1) {
 24        thread_block(TASK_BLOCKED);
 25        //执行 hlt 时必须要保证目前处在开中断的情况下
 26        asm volatile ("sti; hlt" : : : "memory");
 27     }
 28 }
…略
123     /* 实现任务调度 */
124     void schedule() {
…略
138     /* 如果就绪队列中没有可运行的任务，就唤醒 idle */
139     if (list_empty(&thread_ready_list)) {
140         thread_unblock(idle_thread);
141     }
…略
154 }
```

```
…略
183     /* 主动让出cpu，换其他线程运行 */
184     void thread_yield(void) {
185     struct task_struct* cur = running_thread();
186     enum intr_status old_status = intr_disable();
187     ASSERT(!elem_find(&thread_ready_list, &cur->general_tag));
188     list_append(&thread_ready_list, &cur->general_tag);
189     cur->status = TASK_READY;
190     schedule();
191     intr_set_status(old_status);
192 }
193
194 /* 初始化线程环境 */
195 void thread_init(void) {
196     put_str("thread_init start\n");
197
198     list_init(&thread_ready_list);
199     list_init(&thread_all_list);
200     lock_init(&pid_lock);
201
202 /* 将当前main函数创建为线程 */
203     make_main_thread();
204
205     /* 创建idle线程 */
206     idle_thread = thread_start("idle", 10, idle, NULL);
207
208     put_str("thread_init done\n");
209 }
```

我们先看 thread_yield 的实现，它定义在第 184 行，原理还是蛮简单的，核心就是 3 步。

（1）先将当前任务重新加入到就绪队列（队尾），如第 188 行代码。

（2）然后将当前任务的 status 置为 TASK_READY，如第 189 行代码。

（3）最后调用 schedule 重新调度新任务，如第 190 行代码。

值得注意的是前 2 步必须是原子操作，您看，如果在开中断的情况下，刚完成第（1）步把当前任务添加到就绪队列，第（2）步修改状态的代码“cur->status = TASK_READY”尚未执行，因此当前任务的状态依然是 TASK_RUNNING。如果此时发生时钟中断，当前任务正巧被换下 CPU，调度器 schedule 判断当前任务的状态是 TASK_RUNNING，就要将其重新添加到就绪队列，为避免重复添加，这时会触发“ASSERT(!elem_find(&thread_ready_list,&cur->general_tag));”。好了，thread_yield 介绍完了。

一直以来我们的系统有个明显的缺陷（哈哈，好吧，缺陷很多，我不要谦虚了），当就绪队列中没有任务时，调度器没有任务可调度，系统就会通过“ASSERT(!list_empty(&thread_ready_list));”悬停。这种情况不能再持续下去了，所以本次在 thread.c 中顺便塞了个 idle 线程，idle 线程用于系统空闲时，也就是就绪队列中没有任务时才运行的。

咱们看下代码的开头，定义了全局变量 idle_thread，它就是 idle 线程，在第 206 行创建 idle 线程时会为其初始化。

在第 22 行的就是 idle 函数的实现，其原理是在函数体中执行“thread_block(TASK_BLOCKED)”阻塞自己，在其被唤醒后，通过内联汇编执行 hlt 指令，使系统挂起，达到真正的“空闲”。hlt 指令的功能让处理器停止执行指令，也就是将处理器挂起（并不是类似“jmp $”那样空兜 CPU，CPU 利用率 100%），使处理器得到休息，CPU 利用率一下子就掉下来了，在那一小段时间 CPU 利用率为 0。处理器已经停止运行，因此并不会再产生内部异常，唯一能唤醒处理器的就是外部中断，当外部发生后，处理器恢复执行后面的指令。处理器需要被唤醒，必须要保证在开中断的情况下执行 hlt，因此内联汇编代码中，先执行 sti，再执行 hlt。顺便说一下，idle_thread 在第一次创建时会被加入到就绪队列，因此会执行一次，然后阻塞。

当就绪队列为空时，schedule 会在第 140 行将 idle_thread 解除阻塞，也就是唤醒 idle_thread，idle_thread 会执行“sti；hlt”，先开中断，再挂起 CPU。

idle_thread 的创建工作是放在 thread_init 中完成的，即第 206 行。

好啦，有关 thread.c 中的改进就到这，下节继续出发。

13.2.3　实现简单的休眠函数

任何一件工作都建立在一定的基础之上，没有绝对独立的个体，咱们的硬盘中断处理程序也依赖一些基础函数，在继续之前，咱们先去把它们完成再说。

硬盘和 CPU 是相互独立的个体，它们各自并行执行，但由于硬盘是低速设备，其在处理请求时往往消耗很长的时间（不过手册上说最慢的情况也能在 31 秒之内完成），为避免浪费 CPU 资源，在等待硬盘操作的过程中最好把 CPU 主动让出来，让 CPU 去执行其他任务，为实现这种“明智”的行为，我们在 timer.c 中定义休眠函数，当然这只是简易版，精度不是很高，能达到目的就可以了。请见代码 13-6。

代码 13-6　（project/c13/c/device/timer.c）

```
…略
 8 #define IRQ0_FREQUENCY    100
…略
17 #define mil_seconds_per_intr (1000 / IRQ0_FREQUENCY)
18
19 uint32_t ticks;         // ticks 是内核自中断开启以来总共的嘀嗒数
…略
51 /* 以 tick 为单位的 sleep，任何时间形式的 sleep 会转换此 ticks 形式 */
52 static void ticks_to_sleep(uint32_t sleep_ticks) {
53     uint32_t start_tick = ticks;
54
55     /* 若间隔的 ticks 数不够便让出 cpu */
56     while (ticks - start_tick < sleep_ticks) {
57        thread_yield();
58     }
59 }
60
61 /* 以毫秒为单位的 sleep   1 秒= 1000 毫秒 */
62 void mtime_sleep(uint32_t m_seconds) {
63    uint32_t sleep_ticks = DIV_ROUND_UP(m_seconds, mil_seconds_per_intr);
64    ASSERT(sleep_ticks > 0);
65    ticks_to_sleep(sleep_ticks);
66 }
…略
```

程序开头定义的宏 mil_seconds_per_intr，其意义是每多少毫秒发生一次中断，也就是以毫秒计算的中断周期。我们已经把时钟中断频率设置成了每秒 100 次，因此 mil_seconds_per_intr 的值是 1000/100=10 毫秒，1 个中断周期是 10 毫秒，我们用它实现简单的延时功能。

定义在第 52 行的函数是 ticks_to_sleep，它接受一个参数 sleep_ticks，sleep_ticks 是中断发生的次数 ticks，即嘀哒数，功能是让任务休眠 sleep_ticks 个 ticks，也就是此函数按照时钟嘀嗒数来休眠。原理很简单，是利用两次时钟中断发生的“间隔 ticks”实现的，也就是两次采样 ticks 之差。函数中使用的变量 ticks 是全局变量，它是由时钟中断处理函数 intr_timer_handler 更新的，每次时钟中断发生它的值就加 1。咱们这里取两次采样间隔，通过 while 循环不断获取当前的 ticks，只要当前的 ticks 值减去第一次调用时的 ticks（这里是 start_tick），所得的差小于 sleep_ticks（休眠的 ticks 数），就调用 thread_yield 让出 CPU，直到不满足此条件为止（发生了足够多次数的时钟中断），从而达到了延时的目的。

函数 mtime_sleep，其接受一个参数毫秒 m_seconds，其功能是使程序休眠 m_seconds 毫秒，此函数按照毫秒来休眠。然而 mtime_sleep 只是个外壳，咱们并没有真正做到按时间来休眠，按时间休眠的原理是：将休眠的毫秒时间 m_seconds 转换为时钟中断发生的“间隔 ticks 数”，然后调用 ticks_to_sleep，也就是说最终还是由 ticks_to_sleep 完成休眠。代码第 63 行 sleep_ticks 便是将时间转换后的 ticks 数，本质上就是用“休眠的毫秒数”除以“中断发生的毫秒周期”。最后调用“ticks_to_sleep(sleep_ticks)”实现休眠。

在咱们实际应用中，用 mtime_sleep 这种毫秒级的休眠足够应付了，当然也可以专门再写一个秒级的休眠函数，但没必要。好啦，不管休眠的时间粒度是毫秒还是更大，最终要转换成中断次数，再调用 ticks_to_sleep 去完成功能，大伙儿可以自行尝试。

13.2.4　完善硬盘驱动程序

基础部分算是介绍完了，下面开始介绍硬盘的中断处理函数部分，请见代码 13-7-1。

代码 13-7-1　（project/c13/c/device/ide.c）

```
…略
 48 /* 选择读写的硬盘 */
 49 static void select_disk(struct disk* hd) {
 50     uint8_t reg_device = BIT_DEV_MBS | BIT_DEV_LBA;
 51     if (hd->dev_no == 1) {      // 若是从盘就置 DEV 位为 1
 52         reg_device |= BIT_DEV_DEV;
 53     }
 54     outb(reg_dev(hd->my_channel), reg_device);
 55 }
 56
 57 /* 向硬盘控制器写入起始扇区地址及要读写的扇区数 */
 58 static void select_sector(struct disk* hd, uint32_t lba, uint8_t sec_cnt) {
 59     ASSERT(lba <= max_lba);
 60     struct ide_channel* channel = hd->my_channel;
 61
 62     /* 写入要读写的扇区数*/
 63     outb(reg_sect_cnt(channel), sec_cnt);
                                 // 如果 sec_cnt 为 0，则表示写入 256 个扇区
 64
 65     /* 写入 lba 地址，即扇区号 */
 66     outb(reg_lba_l(channel), lba);
           // lba 地址的低 8 位，不用单独取出低 8 位
           // outb 函数中的汇编指令 outb %b0，%w1 会只用 al

 67     outb(reg_lba_m(channel), lba >> 8);     // lba 地址的 8～15 位
 68     outb(reg_lba_h(channel), lba >> 16);    // lba 地址的 16～23 位
 69
 70     /* 因为 lba 地址的第 24～27 位要存储在 device 寄存器的 0～3 位，
 71      * 无法单独写入这 4 位，所以在此处把 device 寄存器再重新写入一次*/
 72     outb(reg_dev(channel), BIT_DEV_MBS | BIT_DEV_LBA | \
       (hd->dev_no == 1 ? BIT_DEV_DEV : 0) | lba >> 24);
 73 }
 74
 75 /* 向通道 channel 发命令 cmd */
 76 static void cmd_out(struct ide_channel* channel, uint8_t cmd) {
 77 /* 只要向硬盘发出了命令便将此标记置为 true，
    硬盘中断处理程序需要根据它来判断 */
 78     channel->expecting_intr = true;
 79     outb(reg_cmd(channel), cmd);
 80 }
 81
 82 /* 硬盘读入 sec_cnt 个扇区的数据到 buf */
 83 static void read_from_sector(struct disk* hd, void* buf, uint8_t sec_cnt) {
 84     uint32_t size_in_byte;
 85     if (sec_cnt == 0) {
 86     /* 因为 sec_cnt 是 8 位变量，由主调函数将其赋值时，若为 256 则会将最高位的 1 丢掉变为 0 */
 87        size_in_byte = 256 * 512;
 88     } else {
 89        size_in_byte = sec_cnt * 512;
 90     }
 91     insw(reg_data(hd->my_channel), buf, size_in_byte / 2);
 92 }
 93
 94 /* 将 buf 中 sec_cnt 扇区的数据写入硬盘 */
 95 static void write2sector(struct disk* hd, void* buf, uint8_t sec_cnt) {
 96     uint32_t size_in_byte;
 97     if (sec_cnt == 0) {
 98     /* 因为 sec_cnt 是 8 位变量，由主调函数将其赋值时，
       若为 256 则会将最高位的 1 丢掉变为 0 */
 99        size_in_byte = 256 * 512;
100     } else {
101        size_in_byte = sec_cnt * 512;
102     }
103     outsw(reg_data(hd->my_channel), buf, size_in_byte / 2);
104 }
```

```
105
106 /* 等待 30 秒 */
107 static bool busy_wait(struct disk* hd) {
108     struct ide_channel* channel = hd->my_channel;
109     uint16_t time_limit = 30 * 1000;         // 可以等待 30000 毫秒
110     while (time_limit -= 10 >= 0) {
111        if (!(inb(reg_status(channel)) & BIT_STAT_BSY)) {
112           return (inb(reg_status(channel)) & BIT_STAT_DRQ);
113        } else {
114           mtime_sleep(10);                  // 睡眠 10 毫秒
115        }
116     }
117     return false;
118 }
…略
```

函数 select_disk 接受一个参数，硬盘指针 hd，功能是选择待操作的硬盘是主盘或从盘。原理是利用 device 寄存器中的 dev 位，该位为 0 表示是通道中的主盘，为 1 表示是通道的从盘。先用宏 BIT_DEV_MBS | BIT_DEV_LBA 拼凑出 device 的值存入变量 reg_device，再根据 hd->dev_no 的值判断是主盘，还是从盘，若为 1 则表示是从盘，于是再将变量 reg_device 的值加上 BIT_DEV_DEV。最后通过 outb 函数将变量 reg_device 写入硬盘所在通道的 device 寄存器，即"reg_dev(hd->my_channel)"，这样就完成了主盘或从盘的选择。

函数 select_sector 接受 3 个参数，硬盘指针 hd、扇区起始地址 lba、扇区数 sec_cnt，功能是向硬盘控制器写入起始扇区地址及要读写的扇区数。功能分两步实现，第 1 步先在 Sector count 寄存器中写入待读写的扇区数，这是由第 63 行的代码完成的，Sector count 寄存器是 8 位宽度，范围是 0～255，因此当写入该寄存器的值为 0 时，表示 256 个扇区。第 2 步是分别在寄存器 LBA low、LBA mid、LBA high 中写入扇区 LBA 地址的低 8 位、中间 8 位和高 8 位，这是由第 66～68 行代码完成的，LBA 地址共 28 位，第 24～27 位写在 device 寄存器的低 4 位中，因此在第 72 行，重新把 device 寄存器写了一次，保留原来信息的同时，又补充了 LBA 的第 24～27 位。

下一个函数是 cmd_out，它接受 2 个参数，通道 channel 和硬盘操作命令 cmd。函数功能是将通道 channel 发 cmd 命令。发命令的时候要将通道的 expecting_intr 置为 true，这是为硬盘中断处理程序埋下伏笔，表示将来该通道发出的中断信号也许是由此次命令操作引起的，因此此通道正期待来自硬盘的中断。然后再将命令 cmd 写入通道的 cmd 寄存器。

下一个函数是 read_from_sector，它接受 3 个参数，分别是待操作的硬盘 hd、缓冲区 buf、读取的扇区数 sec_cnt，功能是从硬盘 hd 中读入 sec_cnt 个扇区的数据到 buf。此函数内部是调用 insw 来完成硬盘读取的，insw 的参数是字，也就是 2 个字节，因此要将扇区数转换成字后再调用 insw。先转换成字节，再将字节除以 2 就是字了，这里将 sec_cnt 转换为字节时要对 sec_cnt 判断一下，如果 sec_cnt 为 0 的话，这表示 256 个扇区，并不是 0 扇区，因此在第 87 行的 size_in_byte 等于 256*512。否则的话，size_in_byte 的值就等于 sec_cnt * 512。size_in_byte 变为合适的字节后，在第 91 行将 size_in_byte/2 转换为字作为参数调用 insw 完成读取扇区。

下面是函数 write2sector，它接受 3 个参数，硬盘 hd、缓冲区 buf、扇区数 sec_cnt，功能是将 buf 中 sec_cnt 扇区的数据写入硬盘 hd。这里写扇区是调用 outsw 完成的，outsw 的参数也是字，因此也需要将扇区数 sec_cnt 转换为字，原理同 read_from_sector 相同，不再赘述。

函数 busy_wait 接受 1 个参数，硬盘 hd，功能是等待硬盘 30 秒。硬盘是个低速设备，因此在其响应过程中，驱动程序可以让出 CPU 使用权使其他任务得到调度，这就是 busy_wait 的作用。为什么要等待硬盘 30 秒呢？在 ata 手册中有这么一句话："All actions required in this state shall be completed within 31 s"，大概意思是所有的操作都应该在 31 秒内完成，所以我们在 30 秒内等待硬盘响应，若成功则返回 true，否则 false。变量 time_limit 的值是 30*1000 毫秒，即 30 秒，接着通过 while 循环，每次将 time_limit 减 10 毫秒，在第 111 行读取 status 寄存器，通过宏 BIT_STAT_BSY 判断 status 寄存器的 BSY 位是否为 1，如果为 1，则表示硬盘繁忙，这时候就调用 mtime_sleep(10)去休眠 10 毫秒。如果 BSY 位为 0 则表示硬盘不忙，接着在第 112 行再次读取 status 寄存器，返回其 DRQ 位的值，DRQ 位为 1 表示硬盘已经准备好数据了，

可以取出（忘记的话参考图 3-32 status 寄存器）。其实 busy_wait 的意思是忙等待，一般忙等待都是指 CPU 空转（空兜），自旋锁就是忙等待的一种形式。咱们这里虽然用 busy_wait 来命名此功能，但本质上我们更高效一点，毕竟是主动让出了 CPU 使用权，这是由 mtime_sleep 函数中间接调用了 thread_yield 函数实现的。

下面介绍 ide.c 的下半部分，见代码 13-7-2。

代码 13-7-2 （project/c13/c/device/ide.c）

```
…略
120 /* 从硬盘读取 sec_cnt 个扇区到 buf */
121 void ide_read(struct disk* hd, uint32_t lba, void* buf, uint32_t sec_cnt) {
122     ASSERT(lba <= max_lba);
123     ASSERT(sec_cnt > 0);
124     lock_acquire (&hd->my_channel->lock);
125
126 /* 1 先选择操作的硬盘 */
127     select_disk(hd);
128
129     uint32_t secs_op;           // 每次操作的扇区数
130     uint32_t secs_done = 0;     // 已完成的扇区数
131     while(secs_done < sec_cnt) {
132        if ((secs_done + 256) <= sec_cnt) {
133             secs_op = 256;
134        } else {
135             secs_op = sec_cnt - secs_done;
136        }
137
138 /* 2 写入待读入的扇区数和起始扇区号 */
139        select_sector(hd, lba + secs_done, secs_op);
140
141 /* 3 执行的命令写入 reg_cmd 寄存器 */
142        cmd_out(hd->my_channel, CMD_READ_SECTOR); //准备开始读数据
143
144     /*********************   阻塞自己的时机  **********************
145        在硬盘已经开始工作（开始在内部读数据或写数据）后才能阻塞自己，
           现在硬盘已经开始忙了，
146        将自己阻塞，等待硬盘完成读操作后通过中断处理程序唤醒自己*/
147        sema_down(&hd->my_channel->disk_done);
148     *************************************************************/
149
150 /* 4 检测硬盘状态是否可读 */
151        /* 醒来后开始执行下面代码*/
152        if (!busy_wait(hd)) {     // 若失败
153            char error[64];
154            sprintf(error, "%s read sector %d failed!!!!!!\n", hd->name, lba);
155            PANIC(error);
156        }
157
158 /* 5 把数据从硬盘的缓冲区中读出 */
159        read_from_sector(hd, (void*)((uint32_t)buf + secs_done * 512),\
           secs_op);
160        secs_done += secs_op;
161     }
162     lock_release(&hd->my_channel->lock);
163 }
164
165     /* 将 buf 中 sec_cnt 扇区数据写入硬盘 */
166     void ide_write(struct disk* hd, uint32_t lba, void* buf, uint32_t sec_cnt) {
167     ASSERT(lba <= max_lba);
168     ASSERT(sec_cnt > 0);
169     lock_acquire (&hd->my_channel->lock);
170
171     /* 1 先选择操作的硬盘 */
172     select_disk(hd);
173
174     uint32_t secs_op;                 // 每次操作的扇区数
175     uint32_t secs_done = 0;           // 已完成的扇区数
176     while(secs_done < sec_cnt) {
177         if ((secs_done + 256) <= sec_cnt) {
178              secs_op = 256;
179         } else {
```

```
180                 secs_op = sec_cnt - secs_done;
181           }
182
183       /* 2 写入待写入的扇区数和起始扇区号 */
184           select_sector(hd, lba + secs_done, secs_op);
185
186       /* 3 执行的命令写入 reg_cmd 寄存器 */
187           cmd_out(hd->my_channel, CMD_WRITE_SECTOR);
                                                  // 准备开始写数据
188
189       /* 4 检测硬盘状态是否可读 */
190           if (!busy_wait(hd)) {         // 若失败
191                 char error[64];
192                 sprintf(error, "%s write sector %d failed!!!!!!\n", hd->name, lba);
193                 PANIC(error);
194           }
195
196       /* 5 将数据写入硬盘 */
197           write2sector(hd, (void*)((uint32_t)buf + secs_done * 512), secs_op);
198
199           /* 在硬盘响应期间阻塞自己 */
200           sema_down(&hd->my_channel->disk_done);
201           secs_done += secs_op;
202       }
203       /* 醒来后开始释放锁*/
204       lock_release(&hd->my_channel->lock);
205 }
206
207 /* 硬盘中断处理程序 */
208 void intr_hd_handler(uint8_t irq_no) {
209       ASSERT(irq_no == 0x2e || irq_no == 0x2f);
210       uint8_t ch_no = irq_no - 0x2e;
211       struct ide_channel* channel = &channels[ch_no];
212       ASSERT(channel->irq_no == irq_no);
213 /* 不必担心此中断是否对应的是这一次的 expecting_intr,
214  * 每次读写硬盘时会申请锁，从而保证了同步一致性 */
215       if (channel->expecting_intr) {
216            channel->expecting_intr = false;
217            sema_up(&channel->disk_done);
218
219       /* 读取状态寄存器使硬盘控制器认为此次的中断已被处理,
          从而硬盘可以继续执行新的读写 */
220            inb(reg_status(channel));
221       }
222 }
223
224 /* 硬盘数据结构初始化 */
225 void ide_init() {
226       printk("ide_init start\n");
…略
233       /* 处理每个通道上的硬盘 */
234       while (channel_no < channel_cnt) {
257            register_handler(channel->irq_no, intr_hd_handler);
258            channel_no++;           // 下一个 channel
259       }
260       printk("ide_init done\n");
261 }
```

代码 13-7-2 中就介绍了 4 个函数，ide_read，ide_write，intr_hd_handler 和 ide_init。以几个函数基本上就是之前介绍函数的封装。

函数 ide_read 接受 4 个参数，硬盘 hd、扇区地址 lba、缓冲区 buf、扇区数量 sec_cnt，功能是从硬盘 hd 的扇区地址 lba 处读取 sec_cnt 个扇区到 buf。在第 124 行，操作硬盘之前先将硬盘所在的通道上锁，从而保证一次只操作同一通道上的一块硬盘。第 127 行通过 select_disk(hd)选择待操作的硬盘。

在继续之前还有点事要和大伙儿交待。因为读写扇区数端口 0x1f2 及 0x172 是 8 位寄存器，故每次读写最多是 255 个扇区（当写入端口值为 0 时，则写入 256 个扇区），所以当读写的端口数超过 256 时，必须拆分成多次读写操作。每当完成一个扇区的读写后，此寄存器的值便减 1，所以当读写失败时，此端口包括尚未完成的扇区数。好啦，可以继续了。

由于硬盘一次只能操作 256 个扇区，为了相对高效一些，咱们的做法是如果待操作的扇区数 sec_cnt 大于 256，咱们尽量一次操作 256 个扇区，余下不足 256 扇区的部分一次性完成。在第 129 行，secs_op 是指每次硬盘操作的扇区数，secs_done 是操作完成的扇区数。下面通过 while 循环，将 sec_cnt 个扇区按照 256 个分组来操作。第 132～136 行便是按照以上思路分组，每次操作 secs_op 个扇区，secs_op 的值不是 256，就是小于 256 的余数，secs_op 等于 256 的情况在 0 次以上，而小于 256 的情况顶多出现 1 次。确定操作的扇区数 secs_op 后，在第 139 行通过 select_sector 函数选择待操作的扇区地址及个数，之后在第 142 行通过 cmd_out 函数向硬盘发读扇区命令。此时硬盘已经开始工作了，前面说过了，硬盘是低速设备，所以在此期间最好是把 CPU 使用权让出去。出让 CPU 使用权有两种方式，一种是用 thread_block 函数阻塞自己，直到运行条件成熟时再运行，也就是由别人唤醒自己后再继续执行。另一种是用 thread_yield 主动交出 CPU，调度器将所有任务调度一圈之后又会让自己运行，但下次运行时，非常有可能的是运行条件尚未具备，比如此处是硬盘还没有完成操作，驱动程序醒来后也无所事事，还得再将 CPU 让出去，因此这里用 thread_block 阻塞驱动程序自己，直到条件成熟时再运行。这种自我阻塞是通过第 147 行对信号量执行 P 操作完成的，即代码“sema_down(&hd->my_channel->disk_done)”。那何时条件成熟呢？也就是何时被唤醒呢？是这样的，硬盘完成操作后会发中断信号，后面介绍的硬盘中断处理程序 intr_hd_handler 会在该通道上执行“sema_up(&channel->disk_done)”，从而唤醒当前的驱动程序。总之，第 147 行的代码执行过后，当前的驱动程序就会阻塞，在等待硬盘操作完成期间开始睡觉了。

当硬盘完成操作后会主动发中断，对应的中断处理程序会将该通道的信号量 disk_done 执行 V 操作，即代码“sema_up(&channel->disk_done)”，从而唤醒驱动程序。驱动程序醒来之后，在第 152 行开始判断硬盘的状态，这是通过代码“busy_wait(hd)”完成的，如果不出现重大硬件损伤的话，通常情况下这种硬盘操作不会失败，如果失败，八成也不是咱们能解决的，因此就将程序通过 PANIC 悬停。如果成功了，就通过 read_from_sector 函数将扇区数据读入到缓冲区(buf+secs_done * 512)处，随着完成的扇区数 secs_done 越来越多，缓冲区地址也会随之偏移。然后使 secs_done 加上 secs_op，更新 secs_done。扇区都读入后，在第 162 行释放锁。

下面是函数 ide_write，它接受 4 个参数，硬盘 hd、写入硬盘的扇区地址 lba、待写入硬盘的数据所在的地址 buf、待写入的数据以扇区大小为单位的数量 sec_cnt。功能是将 buf 中 sec_cnt 扇区数据写入硬盘 hd 的 lba 扇区。ide_write 同 ide_read 的逻辑是相同的，区别是 ide_write 是将缓冲区 buf 中的数据写到硬盘，阻塞的时机也有所不同。对于读硬盘来说，驱动程序阻塞自己是在硬盘开始读扇区之后，对于写硬盘来说，驱动程序阻塞自己是在硬盘开始写扇区之后。总之，阻塞的时机一定是在硬盘开始真正忙活之后的那段“漫长”的时间里。其他方面同 ide_read 类似，不再赘述。

接着是函数 intr_hd_handler，这是硬盘中断处理程序，参数是中断号 irq_no。此中断处理程序负责两个通道的中断，因此 irq_no 要么等于 0x2e，要么等于 0x2f，它们分别是从片 8259A 的 IRQ14 接口和 IRQ15 接口。由于有两个通道，在第 210 行先获取中断所属的通道号，直接用中断号 irq_no 减去 0x2e，所得的差便是中断通道在通道数组 channels 中的索引值。大多数情况下，硬盘发生中断通常是由之前咱们对其发出了操作命令引起的，这是咱们主动使其发中断的方式，之前咱们也说过了，为避免无法分清中断信号来自同一通道上的哪块硬盘，咱们在主动操作硬盘时申请了通道上的锁，因此，如果通道发生了中断信号，它只会是由最近一次操作的硬盘引起的，并不会是之前某次操作的硬盘发出的。在通过 cmd_out 函数主动向硬盘发号施令时，我们会将通道的 channel->expecting_intr 置为 true，也就是宣称此通道正期待中断的来临，这是给硬盘中断处理程序看的，也就是之前所说的为中断处理函数埋下的“伏笔”。因此，在中断处理程序中要判断 channel->expecting_intr 的值是否为 true，毕竟这种硬盘中断才是我们期待的（而不是其他情况，如硬盘自检出了问题，通过中断发出了某种警告，这种情况咱们暂不处理）。如果 channel->expecting_intr 的值为 true，将其置为 false（置为 true 是 cmd_out 的职责），然后给通道的信号量 disk_done 执行 V 操作，即代码“sema_up(&channel->disk_done);”，这样，阻塞在此信号量上的驱动程序便会醒来。

中断处理完成后，需要显式通知硬盘控制器此次中断已经处理完成，否则硬盘便不会产生新的中断，这也是为了保证数据的有效性和安全性。硬盘控制器的中断在下列情况下会被清掉。

- 读取了 status 寄存器。

- 发出了 reset 命令。
- 或者又向 reg_cmd 写了新的命令。

我们采取第 1 种方法，再读一次 status 寄存器，也就是第 220 行的代码“inb(reg_status(channel));”。

最后在第 257 行，我们把硬盘中断处理程序在 ide_init 中完成注册。至此，我们硬盘驱动算是完成了，本节到此为止，下节我们要做点具体的事实战啦。

13.2.5　获取硬盘信息，扫描分区表

我想大伙儿已经有些迫不及待想试试硬盘驱动程序了，本节该是检验它们的时候了，咱们用两件工作来验证，一是向硬盘发 identify 命令获取硬盘的信息，二是扫描分区表。

identify 命令是 0xec，它用于获取硬盘的参数，不过奇怪的是此命令返回的结果都是以字为单位，并不是字节，这一点要注意。咱们只是来验证驱动程序，因此表 13-8 中只列出了咱们用到的三个参数，更多的参数请见 ata 手册。

表 13-8　　identify 命令获得的返回信息（部分）

字 偏 移 量	描　　述
10～19	硬盘序列号，长度为 20 的字符串
27～46	硬盘型号，长度为 40 的字符串
60～61	可供用户使用的扇区数，长度为 2 的整型

分区表扫描的工作稍微复杂一些，根据“图 13-17 分区布局汇总”，咱们需要以 MBR 引导扇区为入口，遍历所有主分区，然后找到总扩展分区，在其中递归遍历每一个子扩展分区，找出逻辑分区。由于涉及到分区的管理，因此我们得给每个分区起个名字，简单起见，最好咱们借鉴现成的 Linux 设备命名方案。Linux 中所有的设备都在/dev/目录下，硬盘命名规则是[x]d[y][n]，其中只有字母 d 是固定的，其他带中括号的字符都是多选值，下面从左到右介绍各个字符。

x 表示硬盘分类，硬盘有两大类，IDE 磁盘和 SCSI 磁盘。h 代表 IDE 磁盘，s 代表 SCSI 磁盘，故 x 取值为 h 和 s。

d 表示 disk，即磁盘。

y 表示设备号，以区分第几个设备，取值范围是小写字符，其中 a 是第 1 个硬盘，b 是第 2 个硬盘，依次类推。

n 表示分区号，也就是一个硬盘上的第几个分区。分区以数字 1 开始，依次类推。

综上所述，sda 表示第 1 个 SCSI 硬盘，hdc 表示第 3 个 IDE 硬盘，sda1 表示第 1 个 SCSI 硬盘的第 1 个分区，hdc3 表示第 3 个 IDE 硬盘的第 3 个分区。咱们这里统一用 SCSI 硬盘的命名规则来命名虚拟硬盘 hd60M.img 和 hd80M.img。其中 hd60M.img 为 sda，hd80M.img 为 sdb。hd60M.img 是裸盘，没有文件系统和分区，因此咱们只处理 hd80M.img，将其上的主分区占据 sdb[1～4]，逻辑分区占据 sdb[5～]。

好啦，现在上代码，先看 ide.c 的前半部分，见代码 13-8-1。

代码 13-8-1　（project/c13/d/device/ide.c）

```
…略
48 /* 用于记录总扩展分区的起始 lba，初始为 0，partition_scan 时以此为标记 */
49 int32_t ext_lba_base = 0;
50
51 uint8_t p_no = 0, l_no = 0;       // 用来记录硬盘主分区和逻辑分区的下标
52
53 struct list partition_list;      // 分区队列
54
55 /* 构建一个 16 字节大小的结构体，用来存分区表项 */
56 struct partition_table_entry {
57     uint8_t  bootable;            // 是否可引导
58     uint8_t  start_head;          // 起始磁头号
59     uint8_t  start_sec;           // 起始扇区号
60     uint8_t  start_chs;           // 起始柱面号
```

```
 61     uint8_t  fs_type;            // 分区类型
 62     uint8_t  end_head;           // 结束磁头号
 63     uint8_t  end_sec;            // 结束扇区号
 64     uint8_t  end_chs;            // 结束柱面号
 65 /* 更需要关注的是下面这两项 */
 66     uint32_t start_lba;          // 本分区起始扇区的 lba 地址
 67     uint32_t sec_cnt;            // 本分区的扇区数目
 68 } __attribute__ ((packed));      // 保证此结构是 16 字节大小
 69
 70 /* 引导扇区，mbr 或 ebr 所在的扇区 */
 71 struct boot_sector {
 72     uint8_t  other[446];         // 引导代码
 73     struct   partition_table_entry partition_table[4];
                                     // 分区表中有 4 项，共 64 字节
 74     uint16_t signature;          // 启动扇区的结束标志是 0x55,0xaa,
 75 } __attribute__ ((packed));
…略
236     /* 将 dst 中 len 个相邻字节交换位置后存入 buf */
237    s static void swap_pairs_bytes(const char* dst, char* buf, uint32_t len) {
238     uint8_t idx;
239     for (idx = 0; idx < len; idx += 2) {
240         /* buf 中存储 dst 中两相邻元素交换位置后的字符串*/
241         buf[idx + 1] = *dst++;
242         buf[idx]     = *dst++;
243     }
244     buf[idx] = '\0';
245 }
246
247 /* 获得硬盘参数信息 */
248 static void identify_disk(struct disk* hd) {
249     char id_info[512];
250     select_disk(hd);
251     cmd_out(hd->my_channel, CMD_IDENTIFY);
252 /* 向硬盘发送指令后便通过信号量阻塞自己，
253   * 待硬盘处理完成后，通过中断处理程序将自己唤醒 */
254     sema_down(&hd->my_channel->disk_done);
255
256 /* 醒来后开始执行下面代码*/
257     if (!busy_wait(hd)) {    //  若失败
258         char error[64];
259         sprintf(error, "%s identify failed!!!!!!\n", hd->name);
260         PANIC(error);
261     }
262     read_from_sector(hd, id_info, 1);
263
264     char buf[64];
265     uint8_t sn_start = 10 * 2, sn_len = 20, md_start = 27 * 2, md_len = 40;
266     swap_pairs_bytes(&id_info[sn_start], buf, sn_len);
267     printk("   disk %s info:\n     SN: %s\n", hd->name, buf);
268     memset(buf, 0, sizeof(buf));
269     swap_pairs_bytes(&id_info[md_start], buf, md_len);
270     printk("     MODULE: %s\n", buf);
271     uint32_t sectors = *(uint32_t*)&id_info[60 * 2];
272     printk("     SECTORS: %d\n", sectors);
273     printk("     CAPACITY: %dMB\n", sectors * 512 / 1024 / 1024);
274 }
```

代码开头先定义了一些数据，挑重点说一下。ext_lba_base 是用在分区表扫描函数 partition_scan 中的，此变量有两个作用，一是作为扫描分区表的标记，partition_scan 若发现 ext_lba_base 为 0 便知道这是第一次扫描，因此初始为 0。另外就是用于记录总扩展分区地址，那时肯定就不为 0 了。partition_list 是所有分区的列表。

接下来的 partition_table_entry 是分区表项，即分区表中的每个分区项，它是按照“表 13-1 分区表项结构”定义的。在结构定义结束处有个“__attribute__ ((packed))”，这是编译器 gcc 提供的属性定义，__attribute__是 gcc 特有的关键字，用于告诉 gcc 在编译时需要做些“特殊处理”，packed 就是“特殊处理”，意为压缩的，即不允许编译器为对齐而在此结构中填充空隙，从而保证结构 partition_table_entry 的大小是 16 字节，这与分区表项的大小是吻合的。

下一个是引导扇区结构体 boot_sector，成员 other 大小是 446 字节，其内容是引导代码，但这并

不重要，它在这里只是用来占位，因为在引导扇区中偏移 446 字节的地方才是分区表，目的是让下面的分区表 partition_table 位置正确，partition_table 是个数组，数组元素是“struct partition_table_entry”，共 4 个元素，即总大小 64 字节。signature 是魔数，类型是 uint16_t，即大小是 2 字节，它是启动扇区的结束标志 0x55，0xaa，占两个字节，最后一个字节是 0xaa。由于 x86 是小端字节序，故此处变量的实际值为 0xaa55。这三个成员加起来总共大小是 512 字节，最后也用“__attribute__ ((packed))”严格保证 512 字节大小。

也许您会疑问，为什么以上两个结构要严格限制大小？因为我们要用它们来读入严格大小的数据，这样方便编程，一会您就知道了。

函数 swap_pairs_bytes 接受 3 个参数，目标数据地址 dst、缓冲区 buf、数据长度 len，功能是将 dst 中 len 个相邻字节交换位置后存入 buf，buf 是 dst 最终转换的结果。此函数用来处理 identify 命令的返回信息，硬盘参数信息是以字为单位的，包括偏移、长度的单位都是字，在这 16 位的字中，相邻字符的位置是互换的，所以通过此函数做转换。

identify_disk 函数接受 1 个参数，硬盘 hd。功能是向硬盘发送 identify 命令以获得硬盘参数信息。函数体中定义了数组 id_info，用来存储向硬盘发送 identify 命令后返回的硬盘参数。

第 250 行先通过 select_disk(hd)选择硬盘，接着通过 cmd_out 函数向硬盘发送了 CMD_IDENTIFY 命令后，此时硬盘开始工作，然后调用 sema_down 阻塞自己。待当前任务被唤醒后，调用“busy_wait(hd)”判断硬盘状态，如果成功了，调用 read_from_sector 从硬盘获取信息到 id_info。

此时 id_info 中已经是硬盘的参数信息了，接下来开始打印它们。第 264 行的数组 buf 是缓冲区，是给 swap_pairs_bytes 使用的，用于存储转换的结果。第 265 行的 sn_start 表示序列号起始字节地址，其值为 10 * 2，10 表示字偏移量，可见表 13-2。md_start 表示型号起始字节地址，其值为 27 * 2，27 表示字偏移量。调用 swap_pairs_bytes 函数后，buf 中已经是字节两两交换的结果，接着在第 267 行输出序列号，后面的输出同理。

下面看 ide.c 的下半部分，见代码 13-8-2。

代码 13-8-2　（project/c13/d/device/ide.c）

```
…略
276 /* 扫描硬盘 hd 中地址为 ext_lba 的扇区中的所有分区 */
277 static void partition_scan(struct disk* hd, uint32_t ext_lba) {
278     struct boot_sector* bs = sys_malloc(sizeof(struct boot_sector));
279     ide_read(hd, ext_lba, bs, 1);
280     uint8_t part_idx = 0;
281     struct partition_table_entry* p = bs->partition_table;
282
283     /* 遍历分区表 4 个分区表项 */
284     while (part_idx++ < 4) {
285         if (p->fs_type == 0x5) {   // 若为扩展分区
286             if (ext_lba_base != 0) {
287    /* 子扩展分区的 start_lba 是相对于主引导扇区中的总扩展分区地址 */
288                  partition_scan(hd, p->start_lba + ext_lba_base);
289             } else {
       // ext_lba_base 为 0 表示是第一次读取引导块，也就是主引导记录所在的扇区

290             /* 记录下扩展分区的起始 lba 地址，
               后面所有的扩展分区地址都相对于此 */
291                 ext_lba_base = p->start_lba;
292                 partition_scan(hd, p->start_lba);
293             }
294         } else if (p->fs_type != 0) {   // 若是有效的分区类型
295             if (ext_lba == 0) {         // 此时全是主分区
296                 hd->prim_parts[p_no].start_lba = ext_lba + p->start_lba;
297                 hd->prim_parts[p_no].sec_cnt = p->sec_cnt;
298                 hd->prim_parts[p_no].my_disk = hd;
299                 list_append(&partition_list, &hd->prim_parts[p_no].part_tag);
300                 sprintf(hd->prim_parts[p_no].name, \
                    "%s%d", hd->name, p_no + 1);
301                 p_no++;
302                 ASSERT(p_no < 4);       // 0,1,2,3
303             } else {
304                 hd->logic_parts[l_no].start_lba = ext_lba + p->start_lba;
305                 hd->logic_parts[l_no].sec_cnt = p->sec_cnt;
```

```
306                 hd->logic_parts[l_no].my_disk = hd;
307                 list_append(&partition_list, &hd->logic_parts[l_no].part_tag);
308                 sprintf(hd->logic_parts[l_no].name, \
                            "%s%d", hd->name, l_no + 5);
                            // 逻辑分区数字从 5 开始，主分区是 1～4
309                 l_no++;
310                 if (l_no >= 8)    // 只支持 8 个逻辑分区，避免数组越界
311                     return;
312             }
313         }
314         p++;
315     }
316     sys_free(bs);
317 }
318
319 /* 打印分区信息 */
320 static bool partition_info(struct list_elem* pelem, int arg UNUSED) {
321     struct partition* part = elem2entry(struct partition, part_tag, pelem);
322     printk("   %s start_lba:0x%x, sec_cnt:0x%x\n",\
        part->name, part->start_lba, part->sec_cnt);
323
324 /* 在此处 return false 与函数本身功能无关，
325  * 只是为了让主调函数 list_traversal 继续向下遍历元素 */
326     return false;
327 }
…略
347     /* 硬盘数据结构初始化 */
348     void ide_init() {
349     printk("ide_init start\n");
…略
383         /* 分别获取两个硬盘的参数及分区信息 */
384         while (dev_no < 2) {
385             struct disk* hd = &channel->devices[dev_no];
386             hd->my_channel = channel;
387             hd->dev_no = dev_no;
388             sprintf(hd->name, "sd%c", 'a' + channel_no * 2 + dev_no);
389             identify_disk(hd);      // 获取硬盘参数
390             if (dev_no != 0) {      // 内核本身的裸硬盘（hd60M.img）不处理
391                 partition_scan(hd, 0);  // 扫描该硬盘上的分区
392             }
393             p_no = 0, l_no = 0;
394             dev_no++;
395         }
396         dev_no = 0;
                // 将硬盘驱动器号置 0，为下一个 channel 的两个硬盘初始化
397         channel_no++;         // 下一个 channel
398     }
399
400     printk("\n   all partition info\n");
401     /* 打印所有分区信息 */
402     list_traversal(&partition_list, partition_info, (int)NULL);
403     printk("ide_init done\n");
404 }
```

函数 partition_scan 接受 2 个参数，硬盘 hd 和扩展扇区地址 ext_lba。功能是扫描硬盘 hd 中地址为 ext_lba 的扇区中的所有分区。

每个子扩展分区中都有 1 个分区表，因此函数 partition_scan 需要针对每一个子扩展分区递归调用，每调用一次，都要用 1 扇区大小的内存来存储子扩展分区所在的扇区，即 MBR 引导扇区或 EBR 引导扇区。注意，由于是递归调用，每次函数未退出时又进行了函数调用，这会导致栈中原函数的局部数据不释放，并且会在栈中生成新的局部变量，尤其是局部变量很大时，这种递归调用会使栈的内存空间消耗量很大，因此用于存储分区表扇区的内存绝对不能用局部变量。比如咱们刚刚在 identify_disk 函数中定义的“char id_info[512]”就是局部变量，局部变量占用的是栈空间，随着子扩展分区的增多，每次调用 partition_scan 扫描分区时都要在栈中占用 512 字节的内存，分区一多，递归调用时栈就会溢出了。咱们的栈加上 PCB 总共是 4096 字节，除了 PCB 外，栈中也就是顶多容纳 7 个扇区，再加上栈中已经用了一部分空间，因此顶多递归 6 次，第 7 次就会使栈溢出。

为避免这种情况，我们在函数体开头第 278 行用 sys_malloc 动态申请 struct boot_sector 大小的内存（1

扇区大小）来存储分区表所在的扇区，返回地址存储在指针 bs。第 279 行通过 ide_read 读入 1 扇区的数据到 bs 指向的内存。之前我们在定义 struct boot_sector 时用“__attribute__ ((packed))”严格限制了结构体大小，因此在第 281 行，可以通过“bs->partition_table”获得分区表地址，并返回给分区表项指针 p。

此时 p 指向分区表数组，在第 284 行起，利用指针 p 遍历所有分区表项。如果分表表项类型 p->fs_type 为 0x5，这说明是扩展分区，意味着要递归调用 partition_scan。前面说过 ext_lba_base 有两个作用，就体现在第 286～293 行的条件判断中。先看第 289 行的 else 分支，它表示 ext_lba_base 为 0，这说明这是第一次调用 partition_scan，此时获取的是 MBR 引导扇区中的分区表，需要记录下总扩展分区的起始 lba 地址，因为后面所有的子扩展分区地址都相对于此。于是在第 291 行用 p->start_lba 为 ext_lba_base 赋值，因此现在 ext_lba_base 是总扩展分区地址，不再为 0。接着在第 292 行用 p->start_lba 即总扩展分区起始地址作为参数继续调用 partition_scan。现在回去看第 286 行，如果 ext_lba_base 不为 0，这说明已不是第 1 次递归调用，此时所获取的分区表是 EBR 引导扇区中的，子扩展分区的起始扇区地址是相对于主引导扇区中的总扩展分区地址 ext_lba_base，因此下面第 288 行用“p->start_lba + ext_lba_base”作为 partition_scan 的参数继续调用。

以上代码片段是处理扩展分区的情况，在第 294 行之后便是处理主分区或逻辑分区。分区类型（文件系统类型）若为 0，则表示 empty，即无效的分区类型，因此在第 294 行判断，只要分区类型 p->fs_type 不等于 0，就认为是有效的分区。如果 partition_scan 的参数 ext_lba 为 0，说明当前是 MBR 引导分区，因此此时的分区表中除了主分区就是总扩展分区，扩展分区已经在上面代码片段中处理过了，因此此时的分区必然是主分区。接着在第 296～298 行将主分区的信息收录到硬盘 hd 的 prim_parts 数组中。第 299 行将分区加入到分区列表 partition_list 中。第 300 行，通过 sprintf 函数拼接字符串为主分区命名，主分区名称从 1 起，如 sda1。第 303 行之后是处理逻辑分区的代码，同主分区类似，不再赘述。

下个函数是 partition_info，它的功能是打印分区信息，此函数被用在 list_traversal 中作为回调函数调用，必须有 2 个参数，但我们只用到分区标记 pelem，所以第 2 个参数 arg 我们用 UNUSED 来修饰，表示未使用。UNUSED 是个宏，定义在 global.h 中：“#define UNUSED __attribute__ ((unused))”，也是利用 gcc 提供的属性 unused 实现的。函数实现很简单，不说了。

接下来要把函数 identify_disk 和 partition_scan 加入到 ide_init 中，这分别是第 389 行和第 391 行完成的。接着在第 402 行调用 list_traversal 打印所有分区信息。

编译运行，结果如图 13-24 所示。

```
Bochs x86 emulator, http://bochs.sourceforge.net/
tss_init start
tss_init and ltr done
syscall_init start
syscall_init done
ide_init start
   disk sda info:
      SN: BXHD00011
      MODULE: Generic 1234
      SECTORS: 121968
      CAPACITY: 59MB
   disk sdb info:
      SN: BXHD00012
      MODULE: Generic 1234
      SECTORS: 163296
      CAPACITY: 79MB

   all partition info
   sdb1 start_lba:0x3F, sec_cnt:0x7DC1
   sdb5 start_lba:0x7E3F, sec_cnt:0x46A1
   sdb6 start_lba:0xC51F, sec_cnt:0x6231
   sdb7 start_lba:0x1278F, sec_cnt:0x3AD1
   sdb8 start_lba:0x1629F, sec_cnt:0x75E1
   sdb9 start_lba:0x1D8BF, sec_cnt:0xA521
ide_init done
CTRL + 3rd button enables mouse
```

▲图 13-24　硬盘及分区信息

大伙儿可以和图 13-24 中的分区信息对比一下，结果是一样的。

好啦，经过漫长的前期工作，驱动程序算是完成了，下一章咱们开始文件系统的实现。

第 14 章　文件系统

本章开始咱们要实现文件系统，这涉及到文件描述符、i 结点等概念，内容较多，工作量较大，难度较高……工作量有点大倒是真的，难度没有，对大家来说只是体力活。待文件系统完成后，咱们就可以在磁盘上加载用户程序了。

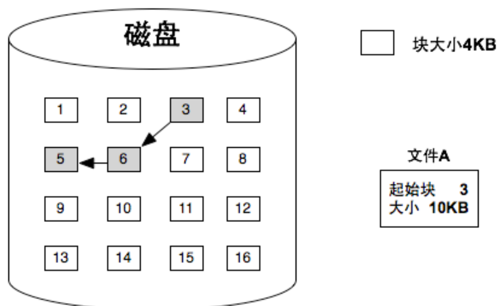
14.1 文件系统概念简介

在实现文件系统之前，咱们要介绍文件系统中几个重要的概念。咱们先了解下 Linux 中是怎么管理文件的。尽管是参照 Linux 文件系统，但不可能是原样照搬过来，否则您直接看 Linux 源码不就行了^_^。咱们的目的是用最少的代码介绍清楚内核的原理，只要了解操作系统的核心思路就行了。再说，Linux 太庞大了，小弟不可能凭一已之力全部消化，而且作为一个成熟的操作系统，Linux 的强大是在管理方面，管理策略占了整个系统的 95%以上，而咱们只做 5%以内的事。

14.1.1　inode、间接块索引表、文件控制块 FCB 简介

硬盘是低速设备，其读写单位是扇区，为了避免频繁访问硬盘，操作系统不会有了一扇区数据就去读写一次磁盘，往往等数据积攒到“足够大小”时才一次性访问硬盘，这足够大小的数据就是块，硬盘读写单位是扇区，因此一个块是由多个扇区组成的，块大小是扇区大小的整数倍。在 Windows 中，块被称为簇，比如在 Windows 中格式化分区时，若选择文件系统类型为 FAT32，我们还可以选择多种不同大小的簇，有 4KB、32KB 等。以下为叙述方便，一律统称为块。块是文件系统的读写单位，因此文件至少要占据一个块，当文件体积大于 1 个块时，文件肯定被拆分成多个块来存储，那么问题来了，这多个块该如何组织到一起？现在要讨论文件的组织方式了。拿 FAT 文件系统来说，FAT 称为文件分配表，在此文件系统中存储的文件，其所有的块被用于链式结构来组织，在每个块的最后存储下一个块的地址，从而块与块之间串联到一起，这样一来，文件可以不连续存储，文件中的块可以分布在各个零散的空间中，有效地利用了存储空间，或者说是提升了磁盘的利用率，相当于节省了空间。其结构如图 14-1 所示。

图中文件 A 大小是 10KB，块大小是 4KB，因此要占用 3 个块，最后一个块（块 5）中浪费了 2KB 空间。但通过链式结构来组织文件的弊端是当访问文件中的某个块时，必须要从头开始遍历块结点，比如要访问文件 A 的第 3 个块（块 5），需要先访问第 1 个块（块 3），获得第 2 块的地址（块 6），再从第 2 块中获取第 3 块（块 5）的地址，软件上算法效率低下，而且每访问一个结点，就要涉及一次硬盘寻道，使得对原本低速的设备访问更加频繁了。也许连微软自己也受不了它了，后来推出了 NTFS 文件系统。



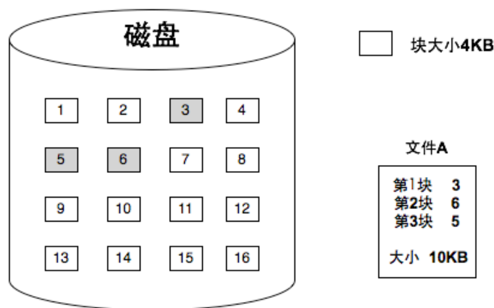
▲图 14-1　文件的链式组织结构

以上是文件用链式结构组织的形式，注意啦，文件组织形式是对各个文件而言的，FAT 文件系统中每个文件都有这么个单独的链式结构来组织、跟踪文件的所有块。下面讨论下另一种文件组织形式，UINX 操作系统中的索引结构——inode，在咱们的实现中也以 inode 为主，因此下面重点说下。

UNIX 文件系统比较先进，它将文件以索引结构来组织，避免了访问某一数据块需要从头把其前所有数据块再遍

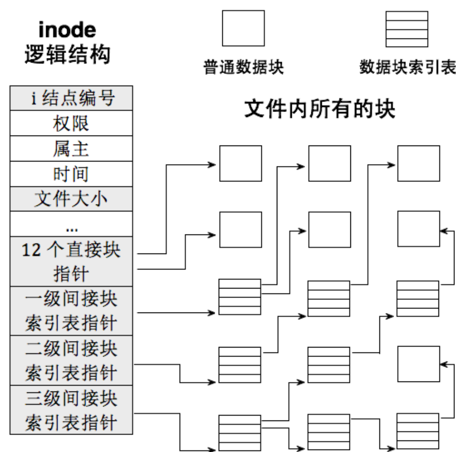
历一次的缺点。采用索引结构的文件系统，文件中的块依然可以分散到不连续的零散空间中，保留了磁盘高利用率的优点，更重要的是文件系统为每个文件的所有块建立了一个索引表，索引表就是块地址数组，每个数组元素就是块的地址，数组元素下标是文件块的索引，第 n 个数组元素指向文件中的第 n 个块，这样访问任意一个块的时候，只要从索引表中获得块地址就可以了，速度大大提升。包含此索引表的索引结构称为 inode，即 index node，索引结点，用来索引、跟踪一个文件的所有块。强调下，inode 是文件索引结构组织形式的具体体现，必须为每个文件都单独配备一个这样的元信息数据结构，因此在 UINX 文件系统中，一个文件必须对应一个 inode，磁盘中有多少文件就有多少 inode。

采用索引结构存储文件 A 的逻辑示意图如图 14-2 所示。



▲图 14-2　文件的索引组织结构

用索引结构的缺点是索引表本身要占用一定的存储空间，文件要是很大时，块就比较多，索引表项就要跟着增多，难道要让索引表就变得很大吗？这显然不科学，UNIX 为解决这个问题采取了折中的方法，将一部分块放在索引表中，如果文件很大，将其他块放在另一个索引表，具体做法是：每个索引表中共 15 个索引项，暂时称此索引表为老索引表。老索引表中前 12 个索引项是文件的前 12 个块的地址，它们是文件的直接块，即可直接获得地址的块。若文件大于 12 个块，那就再建立个新的块索引表，新索引表称为一级间接块索引表，表中可容纳 256 个块的地址，各表项都是块的地址，这 256 个块地址需要通过一级间接块索引表才能获得，因此称为“间接块”，这也是一级间接块索引表中包含“间接”二字的原因。此表也要占用一个物理块来存储，该物理块的地址存储到老索引表的第 13 个索引项中。有了一级间接块索引表，文件最大可达 12+256=268 个块。有同学说了，要是文件超过 268 个块怎么办？这个好办，我们可以再建立二级间接块索引表，此表中各表项存储的是一级间接块索引表，然后在老索引表中第 14 个索引项存储二级间接块索引表所在块的地址。有了二级间接块索引表，文件最大可达（12+256+256*256）个块，自己算算多少块吧。再不够的话，可以再建立三级间接块索引表，表中各表项存储的是二级间接块索引表，然后在二级间接块索引表中建立一级间接块索引表，三级间接块索引表所在块的地址记录在老索引表的第 15 个索引项中。有了三级间接块索引表，文件最大可达（12+256+256*256+256*256*256）个块，不用再算了，总之很大。如果超过了这个限度，那就没办法了，正常情况下也不会出现这种情况，真要是有这样的超大文件出现，那就只能用 mv 命令将其切割成多个小文件了。inode 的逻辑结构如图 14-3 所示。



▲图 14-3　inode 结点结构与各级间接块索引表

您看，在 inode 结构中，几乎囊括了一个文件的所有信息，i 结点编号是指此 inode 的序号，这通常是指它在 inode 数组中的下标（后面会讲到）。权限是指读、写、执行。属主是指文件的拥有者，时间是指创建时间、修改时间、访问时间等。文件大小是指文件的字节尺寸。下面这些连续的各种块指针及索引表指针是文件所有块的索引，也就是指向文件的实体部分。文件系统为实现文件管理方案，必然会创造出一些辅助管理的数据结构，只要用于管理、控制文件相关信息的数据结构都被称为 FCB（File Contrl Block），即文件控制块，inode 也是这种结构，因此 inode 是 FCB 的一种。形象一点地说 inode 相当于通往文件实体数据块的大门，这一点的作用类似于内存段的段描述符，只不过 inode 是文件实体数据块的描述符，里面规定了访问此文件数据块的权限、属主等安全方面的条件，有文件数据块总大小等描述信息，有文件实体数据块的具体地址。总之我想强调的是 inode 是文件在文件系统上的元信息（文件本身的元信息是它自己的文件头），要想通过文件系统获得文件的实体，必须先要找到文件的 inode，从这个意义上来说，inode 等同于文件。另外，咱们并没打算实现用户权限管理，因此不会完整地实现这些属性，目前只要把灰色的部分完成就可以了。

Linux 是后起之秀，它的文件系统借鉴了 inode 结构，同样是一个文件具有一个 inode，有多少文件就有多少 inode。但是硬盘空间是有限的，而且 inode 本身也要占用磁盘空间来存储，文件系统是针对各个分区来管理磁盘空间的，因此，各个分区的可用空间实际上被所有文件的 inode 结构和所有文件的数据块共享（只是暂且先这么说，因为还有超级块、各种位图、目录项等也占用磁盘空间）。inode 结构是固定的，其大小自然也是固定的，现假设“所有文件的 inode 结构”使用的磁盘空间为 x，“所有文件的数据块”使用的磁盘空间为 y，那么 x+y=分区容量。您看，分区容量大小是固定的，x 和 y 是变量，都是一方决定另一方占用的空间大小，这样在为分区规划文件系统的元信息时，必须要先将一方确定下来才行。这个先被确定下来的变量是 x，x=文件数*inode 大小，因此分区最大创建的文件数是有限制的（实际上我们用 mkfs 命令分区时也是这样做的，先确定 inode 数量），正如 Linux 中每分区的 inode 数量是固定的，可以用 tune2fs 命令查看 inode 数量。

inode 的数量等于文件的数量，为方便管理，分区中所有文件的 inode 通过一个大表格来维护，此表格称为 inode_table，在计算机中表格都可以用数组来表示，因此 inode_table 本质上就是 inode 数组，数组元素的下标便是文件 inode 的编号。

文件的数量间接决定了分区空间的利用率，说“间接”的原因是文件可大可小，每个文件大小不一，比如对于大小分别为 10KB 和 10GB 的文件，同一个 16GB 大小的分区可分别容纳的文件数必然相差甚远。因此一个分区的利用率分为 inode 的利用率和磁盘空间利用率两种，在 Linux 中可以通过 df –i 命令查看 inode 利用率，不加参数执行 df 时，查看的是空间利用率。

好啦，有关 inode 就介绍到这，如果感觉模糊的话，等咱们在实践上再加深理解吧。

14.1.2 目录项与目录简介

经过我反复强调，我想大伙儿已经了解了每个文件都要有一个 inode，inode 中记录了文件的大部分信息，其地位等同于文件。但如此重要的数据结构中却少了一个对用户来说最重要的属性。想想看，我们是通过文件名来访问文件的，inode 中却没有文件名，其实可以在 inode 中存储文件名，只是如果这样做的话，需要改变文件系统的设计，而且也不是现代文件系统的做法，原因是文件系统对文件是用 inode 来描述的，只要给出 inode，文件系统便能够找到文件实体的数据块，因此文件名对操作系统（文件系统）来说并不重要，从这也能够看出，inode 是文件系统需要的东西，并不是给用户准备的（用户无法直接使用 inode，只能间接用到）。可是文件系统毕竟是人创造的，创建文件系统的目的是为了帮助人们解决问题，而不是制造麻烦，文件系统的用户是人，人是通过文件名来与文件打交道的，不可能要让用户记住文件的 inode 编号。那么问题来了，文件系统是如何把文件名和 inode 关联到一起的呢？

再想想看，文件名在查找文件的时候才用到，此处的“查找文件”并不是狭义上的用搜索或 find 命令查找文件，而是广义上任何需要用到文件名的情况，无论文件名是作为可执行程序，或是作为参数，这都属于查找文件的范畴。无论文件在哪个路径，它肯定要位于某个目录中（至少要有个根目录'/'），因此文件名应该存储在和目录相关的地方，这就引出了另外一个概念——目录。什么是目录？

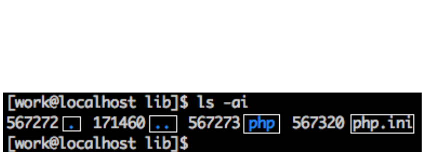
在 Linux 中，目录和文件都用 inode 来表示，因此目录也是文件，只是目录是包含文件的文件。为了表述清楚这两种文件，我们这里称目录为目录文件，一般意义上的文件称为普通文件。在 Linux 中，只要是文件就一定会有个与之匹配的 inode。按理说，既然同一种 inode 同时表示目录和普通文件，这说明在 inode 中它们是没区别的，那文件系统是如何区分目录和文件呢？在磁盘上的文件系统中（注意啦，我说的是磁盘上），没有一种专门称为目录的数据结构，磁盘上有的只是 inode，这是我一直反复强调 inode 重要性的原因之一。inode 用于描述一个文件实体的数据块，至于该数据块中记录的是什么，这并不是 inode 决定的，inode 也不必关心它是什么。既然同一种 inode 既用来表示普通文件，又用来表示目录文件，inode 结构相同，因此区分该 inode 是普通文件，还是目录文件，唯一的地方只能是数据块本身的内容了，如您所料，它们必须是不同的。如果该 inode 表示的是普通文件，此 inode 指向的数据块中的内容应该是普通文件自己的数据。如果该 inode 表示的是目录文件，此 inode 指向的数据块中的内容应该是该目录下的目录项，咱们只会支持目录文件和普通文件（不会存在管道文件、socket 之类），因此目录中要么是普通文件的目录项，要么是目录文件的目录项。

什么目录项？其实咱们每天都要见到它们，先来个亲民一些的介绍。

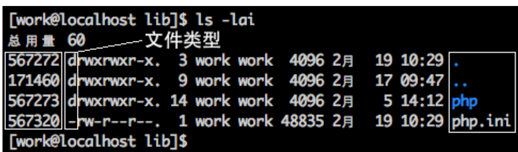
不管文件是普通文件，还是目录文件，它总会存在于某个目录中，所有的普通文件或目录文件都存在于根目录'/'之下，根目录是所有目录的父目录。我们在某个目录下执行 ls 命令的时候，输出的结果就是目录项的外在展现，如图 14-4 所示。

图 14-4 中输出的是在当前目录 lib 下执行 ls 后的结果，参数 a 显示隐藏文件，为的是显示'. '和'.. '这两个文件，参数 i 是为了显示 inode 编号。您看到的框框中 4 个文件，三个目录和一个普通文件（php.ini），它们的名字就是 lib 目录中各条目中记录的名字，这个条目称为目录项。

有时候我们在执行 ls 命令时还能看到文件属主、权限等信息，那是由于加了参数 l（命令 ll 是 ls 别名：'ls -l --color=auto'）。不过这些额外的信息是来自 inode，如图 14-5 所示。



▲图 14-4　目录中的目录项



▲图 14-5　目录项信息及 inode 信息

参数 l 用于长列表格式显示文件信息，两边框框中的内容来自目录项，左边第 2 个框框表示文件类型，它也来自于目录项，中间未加框的内容来自 inode。您看，这里的文件类型中，'d'表示文件是目录，'-'表示文件是普通文件。之前说 inode 不关心数据块中记录的是什么，也就是说数据块中存储的是目录，还是普通文件的实体数据，inode 是不知道的。那现在您应该清楚了，inode 不知道的事，目录项知道。

好啦，现在可以想像一下目录项的样子了。目录相当于个文件列表（或者是表格），每个文件在目录中都是一个 entry（条目、项），各个 entry 中的内容包括文件名、文件类型，为了定位文件的数据，entry 中至少还要包括 inode 编号，这个 entry 是目录中各个文件的描述，它称为目录项，目录项中至少要包括文件名、文件类型及文件对应的 inode 编号，还是拿图 14-4 中的 lib 目录来说，其目录项如图 14-6 所示。

目录lib/下的目录项

inode 编号	文件名	文件类型
567272	.	目录
171460	..	目录
567273	php	目录
567320	php.ini	普通文件

▲图 14-6　lib 目录下的目录项

您看，目录项中包含文件名、inode 编号和文件类型，它们三个的作用有两个，一是标识此 Inode 表示的文件是目录，还是普通文件，也就是 inode 所指向数据块中的内容是什么。二是将文件名与 inode 做个绑定关联，这样用户便可以通过文件名来找到文件的实体数据。

有了目录项后，通过文件名找文件实体数据块的流程是。

（1）在目录中找到文件名所在的目录项。

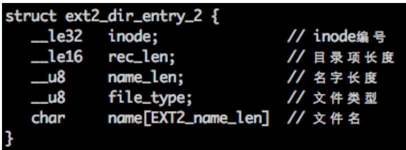
（2）从目录项中获取 inode 编号。

（3）用 inode 编号作为 inode 数组的索引下标，找到 inode。

（4）从该 inode 中获取数据块的地址，读取数据块。

以上是一个目录项中最基本的属性，在较成熟文件系统的 inode 中属性会多一些，拿 Linux 的 ext2 文件系统来说，其目录项结构是这样的，如图 14-7 所示。

其中的文件类型 file_type 可能的取值如图 14-8 所示。



▲图 14-7　ext2 目录项



▲图 14-8　ext2 文件类型

只要用于管理文件相关信息的数据结构都可称为文件控制块，因此目录项也是。创建文件的本质是创建了文

件的文件控制块，即目录项和 inode，这一点在与文件创建的相关实践中大伙儿会更有体会。

我估计头一次接触 inode 和目录项的同学可能还是很“晕”，下面总结梳理下以上所介绍的内容。

（1）每个文件都有自己单独的 inode，inode 是文件实体数据块在文件系统上的元信息。

（2）所有文件的 inode 集中管理，形成 inode 数组，每个 inode 的编号就是在该 inode 数组中的下标。

（3）inode 中的前 12 个直接数据块指针和后 3 个间接块索引表用于指向文件的数据块实体。

（4）文件系统中并不存在具体称为“目录”的数据结构，同样也没有称为“普通文件”的数据结构，统一用同一种 inode 表示。inode 表示的文件是普通文件，还是目录文件，取决于 inode 所指向数据块中的实际内容是什么，即数据块中的内容要么是普通文件本身的数据，要么是目录中的目录项。

（5）目录项仅存在于 inode 指向的数据块中，有目录项的数据块就是目录，目录项所属的 inode 指向的所有数据块便是目录。

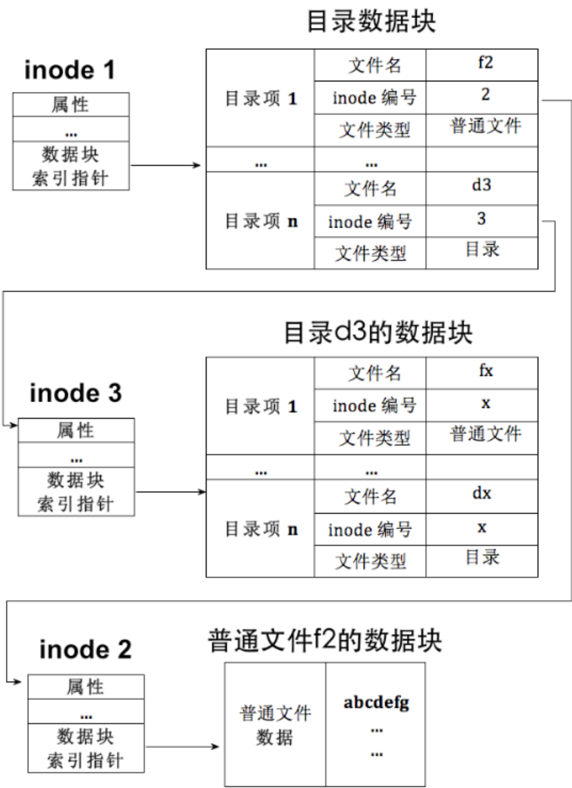
（6）目录项中记录的是文件名、文件 inode 的编号和文件类型，目录项起到的作用有两个，一是粘合文件名及 inode，使文件名和 inode 关联绑定，二是标识此 inode 所指向的数据块中的数据类型（比如是普通文件，还是目录，当然还有更多的类型）。

（7）inode 是文件的“实质”，但它并不能直接引用，必须通过文件名找到文件名所在的目录项，然后从该目录项中获得 inode 的编号，然后用此编号到 inode 数组中去找相关的 inode，最终找到文件的数据块。

inode 和目录项的关系如图 14-9 所示。

看了这张图后，似乎有点清楚了，但仔细一想似乎又感到更“蒙圈”了，您看，要想找到文件（普通文件或目录文件）的数据块，必须找到文件的 inode。inode 之所以被引用（找到），是因为在文件名所在的目录项中有记录它的编号，但是目录项是在目录文件的数据块中，而数据块必须通过 inode 才能找到……寻找过程似乎陷入了死循环。任何看似循环的流程都有个初始，就像心脏的第一次跳动一样。对于这种看似死循环的问题，无外乎的原因就是它的上层目录看似是无休止的，只要有个固定的目录就解决了。也许有同学已经想到了，对，就是那个根目录'/'，它是所有目录的父目录，每个分区都有自己的根目录，创建文件系统之后它的位置就是固定不变的，也就是说，在文件系统的设计中，根目录所在数据块的地址是被“写死”的，查找任意文件时，都直接到根目录的数据块中找相关的目录项，然后递归查找，最终可以找到任意子目录中的文件。

好啦，概念差不多就说到这，其他的还是要留在实践中验证吧，大伙儿下节再见。



▲图 14-9　inode 与目录项的关系

14.1.3　超级块与文件系统布局

到现在为止，我们已经介绍了 inode 和目录项的作用，还顺带说了下根目录，这对于实现文件系统来说差不多了，本节还要给大伙儿补充个重要概念，这就是超级块。

超级块是干吗的呢？大伙儿想想，我们已经知道每个文件都有个 inode，所有的 inode 都放在 inode 数组中，请问，inode 数组在哪里？大小是多少？还有，尽管我已经和大伙儿说过根目录的地址是固定写死的，但每个分区都有自己的根目录，其地址并不统一，也许分区 a 的根目录在本分区的第 500 扇区，分区 b 的根目录在本分区的第 2012 扇区，虽然固定，但不统一。您看，既然各分区根目录的位置并不统一，这就说明该地址必然放在某处保存，在各分区中，该处的地址必然是固定且统一，以备随时读取，总之表

面上看似变化复杂的事物被追溯到根源时都是及其固定简单的，大道至简。为了避免访问越界，根目录也应该有个大小。这就像项目的配置文件，虽然很多东西在一开始都可以被固定下来，但依然要将它保存在配置文件中，对于文件系统也一样，我们需要在某个固定地方去获取文件系统元信息的配置，这个地方就是超级块，超级块是保存文件系统元信息的元信息。

在超级块中保存的信息当然不止这些，这取决于文件系统的复杂性，越复杂的文件系统在超级块中保存的信息就越多，大伙儿可以参考下 ext2 文件系统的超级块，这里就不给大伙儿贴图了，内容挺多的，上面的很多东西我也不懂，看了之后我也解释不了，还是算了。现在咱们讨论下，实现一个最基本的文件系统需要哪些元信息。

现在咱们说说 inode 数组还需要哪些维护信息。前面说过了，文件系统是针对各个分区来管理的，inode 代表文件，因此各分区都有自己的 inode 数组。尽管各分区可创建的最大文件数是固定的，但这并不表示所有分区的最大文件数都相同，各分区可创建的最大文件数是在为分区创建文件系统（格式化）时设置的（如在用 mkfs 工具为分区创建文件系统时就有 inode 数量的设置），我们可以为不同的分区设置不同的文件数。总之，各分区 inode 数组长度是固定的，等于最大文件数。既然 inode 数量是有限的，必须要有一种管理 inode 使用情况的方法，我们已经会用位图管理内存了，因此咱们也用位图来管理 inode 的使用情况，好，现在又多了一个元信息，inode 位图。

除了文件系统的元信息外，就剩下可用的空闲块了，文件系统被创建出来的目的就是为了合理科学地管理这些空闲块，空闲块也是有限的，因此空闲块的使用情况也需要被跟踪，所以咱们也要为这些空闲块准备个位图。

咱们所讨论出来的内容已经足够实现一个简单的文件系统了，总结一下它们有：inode 数组的地址及大小、inode 位图地址及大小、根目录的地址和大小、空闲块位图的地址和大小，以上这几类信息要在超级块中保存。因此一个简单的超级块结构如图 14-10 所示。

超级块

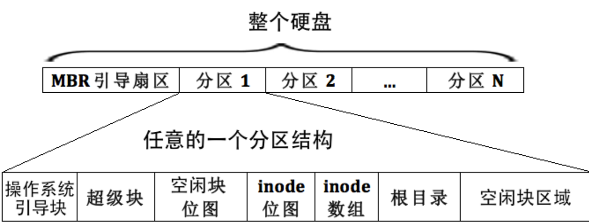
魔数
数据块数量
inode 数量
分区起始扇区地址
空闲块位图地址
空闲块位图大小
inode 位图地址
inode 位图大小
inode 数组地址
inode 数组大小
根目录地址
根目录大小
空闲块起始地址
...

▲图 14-10　超级块逻辑结构

图 14-10 中列出的大部分属性都很容易理解，但不知道大伙儿对“魔数”有没有疑问，它是干吗的？通常情况下魔数用来确定文件系统的类型的标志，用它来区别于其他文件系统。比如某个操作系统支持多种文件系统，通常就是根据此魔数先判断文件系统类型，然后调用不同的文件系统驱动程序访问该分区。因此，访问不同文件系统上的文件，按理说先要了解该文件系统中元信息的意义，然后按照该文件系统的存取流程去操作就可以了（话虽这么说，其实工作量还是蛮大的）。

超级块是文件系统元信息的“配置文件”，它是在为分区创建文件系统时创建的，所有有关文件系统元信息的配置都在超级块中，因此超级块的位置和大小不能再被“配置”了，必须是固定的，它被固定存储在各分区的第 2 个扇区，通常是占用一个扇区的大小，具体大小与实际文件系统类型为准。

对文件系统来说，该有的数据都有了，下面该讨论下它们在磁盘上的布局了。图 14-11 所示是一个典型的 inode 结构的文件系统布局，Linux 早期使用的文件系统就是这样布局的，咱们参考了 ext2 文件系统，ext2 文件系统是一款很成熟强大的文件系统，很多 Linux 发行版的文件系统都基于它。咱们的文件系统布局和 ext2 相比简单了很多，少了一些块组和块组描述符。



▲图 14-11　文件系统布局

图中只是一个分区文件系统元信息的布局，其他各分区中是同构的，咱们也按照这种布局来设计自己的文件系统。操作系统引导块就是曾经介绍过的操作系统引导记录 OBR 所在的地址，即操作系统引导扇区，它位于各分区最开始的扇区，根据文件系统类型的不同，引导程序可能占用多个扇区，这多个扇区组成一个数据块，因此这里标出的是引导“块”，而不是引导“扇区”。在操作系统引导块后面的依次是超级块、空闲块的位图、inode 位图、inode 数组、根目录、空闲块区域。根目录和空闲块区域是真正用于存储数据的区域，除了这两部分，其他几个部分占用的扇区数取决于分区的容量大小，或者是在创建文件系统的过程中手动设置。

随着这张布局图的出现，咱们有关文件系统的理论部分也就告一段落了，后面等待咱们的将是更多的实践，兄弟们咱们一起加油吧。

14.2 创建文件系统

本节咱们将在硬盘 hd80M.img 的各分区上创建文件系统，后续的文件操作都要基于本节的工作，大伙儿走起。

14.2.1 创建超级块、i 结点、目录项

在创建文件系统之前，有一些基础数据结构要先创建，它们是超级块、inode 和目录项，您看，这一节创建三种数据结构，这说明它们都比较简单，相信本节很快搞定。

下面先给大伙儿介绍超级块，早在介绍硬盘驱动时就已经提到过它了，我们甚至在代码中定义了它的指针，不过幸好没有真正用到，否则编译就不会通过。现在把欠大伙儿的超级块奉上。有关文件操作的代码我们定义在 fs 目录下，本节咱们新建这个目录，超级块所在的文件位于 fs/super_block.h 中，见代码 14-1。

代码 14-1 （project/c14/a/fs/super_block.h）

```
…略
 5 /* 超级块 */
 6 struct super_block {
 7     uint32_t magic;
             // 用来标识文件系统类型
             // 支持多文件系统的操作系统通过此标志来识别文件系统类型
 8     uint32_t sec_cnt;                // 本分区总共的扇区数
 9     uint32_t inode_cnt;              // 本分区中 inode 数量
10     uint32_t part_lba_base;          // 本分区的起始 lba 地址
11
12     uint32_t block_bitmap_lba;       // 块位图本身起始扇区地址
13     uint32_t block_bitmap_sects;     // 扇区位图本身占用的扇区数量
14
15     uint32_t inode_bitmap_lba;       // i 结点位图起始扇区 lba 地址
16     uint32_t inode_bitmap_sects;     // i 结点位图占用的扇区数量
17
18     uint32_t inode_table_lba;        // i 结点表起始扇区 lba 地址
19     uint32_t inode_table_sects;      // i 结点表占用的扇区数量
20
21     uint32_t data_start_lba;         // 数据区开始的第一个扇区号
22     uint32_t root_inode_no;          // 根目录所在的 I 结点号
23     uint32_t dir_entry_size;         // 目录项大小
24
25     uint8_t  pad[460];               // 加上 460 字节，凑够 512 字节 1 扇区大小
26 } __attribute__ ((packed));
```

为方便写程序，咱们的数据块大小与扇区大小一致，即 1 块等于 1 扇区，请大伙儿知晓。

咱们的文件系统没那么强大，因此超级块中没什么太多的内容，连 1 扇区都不到，但磁盘操作要以扇区为单位，咱们交给硬盘的数据必须是扇区大小的整数倍。为了凑足 1 扇区，即 512 字节，在超级块的最后定义了 460 字节的 pad 数组，仅仅是用来填充扇区用的。大伙儿可以算一下，pad 之前的所有变量大小之和一定为 52 字节，13 个变量，每个变量 4 字节。为了保证编译后的超级块实例大小为 512 字节，在第 26 行添加了“__attribute__ ((packed));”。

好啦，对于超级块中其他成员的说明，大伙儿自己看看代码中的注释吧，写得还是很清楚的。下面看另一个数据结构，inode，inode 定义在 fs/inode.h 中，见代码 14-2。

代码 14-2 （project/c14/a/fs/inode.h）

```
…略
 6 /* inode 结构 */
 7 struct inode {
 8     uint32_t i_no;           // inode 编号
 9
```

```
10 /* 当此 inode 是文件时，i_size 是指文件大小,
11 若此 inode 是目录，i_size 是指该目录下所有目录项大小之和*/
12     uint32_t i_size;
13
14     uint32_t i_open_cnts;    // 记录此文件被打开的次数
15     bool write_deny;         // 写文件不能并行，进程写文件前检查此标识
16
17 /* i_sectors[0-11]是直接块，i_sectors[12]用来存储一级间接块指针 */
18     uint32_t i_sectors[13];
19     struct list_elem inode_tag;
20 };
21 #endif
```

inode 结构中，i_no 是 inode 编号，它是在 inode 数组中的下标。

i_size 是此 inode 指向的文件的大小，目录也是用 inode 指代，因此当 inode 指向的是普通文件时，i_size 表示普通文件的大小，当 inode 指向的是目录时，i_size 表示目录中所有目录项的大小之和。注意 i_size 是以字节为单位的大小，并不是以数据块为单位，说明一下，为方便编码，咱们的数据块大小等于扇区大小。

i_open_cnts 表示此文件被打开的次数，它在关闭文件时，回收与之相关的资源时使用，后面用到时再讨论。

write_deny 用于限制文件的并行写操作，我们都有这样的常识，不能让多个用户同时写 1 个文件，这样后写入的内容会覆盖之前写入的内容，从而引起数据混乱，因此必须保证文件在执行写操作时，该文件不能再有其他并行的写操作，多个写操作应该以串行的方式进行。当 write_deny 为 true 时表示已经有任务在写该文件了，此文件的其他写操作应该被拒绝。

i_sectors 是数据块的指针，咱们的块大小就是 1 扇区，因此为了不引起迷惑，直接把块数组命名为 i_sector（而不是像 ext2 中的 i_block）。其中，数据的前 12 个块 i_sectors[0-11]是直接块，也就是它们中记录的是数据块的扇区地址，i_sectors[12]用来存储一级间接块索引表的扇区地址，咱们只打算支持一级间接块，不过稍微不同的是扇区大小是 512 字节，并且块地址用 4 字节来表示，因此咱们支持的一级间接块数量是 128 个，即咱们总共支持 128+12=140 个块（扇区）。二级三级同理，只是稍微麻烦了一点，大伙儿有兴趣自行实现吧。

inode_tag 是此 inode 的标识，用于加入“已打开的 inode 列表”，此列表将在以后定义。建立此列表的目的是这样的：由于 inode 是从硬盘上保存的，文件被打开时，肯定是先要从硬盘上载入其 inode，但硬盘比较慢，为了避免下次再打开该文件时还要从硬盘上重复载入 inode，应该在该文件第一次被打开时就将其 inode 加入到内存缓存中，每次打开一个文件时，先在此缓冲中查找相关的 inode，如果有就直接使用，否则再从硬盘上读取 inode，然后再加入此缓存。这个内存缓存就是“已打开的 inode 队列”，用到它的时候咱们再介绍。

下面咱们看目录项的定义，它定义在 fs/dir.h 中，如代码 14-3 所示。

代码 14-3　（project/c14/a/fs/dir.h）

```
…略
 9 #define MAX_FILE_NAME_LEN  16     // 最大文件名长度
10
11 /* 目录结构 */
12 struct dir {
13     struct inode* inode;
14     uint32_t dir_pos;            // 记录在目录内的偏移
15     uint8_t dir_buf[512];        // 目录的数据缓存
16 };
17
18 /* 目录项结构 */
19 struct dir_entry {
20     char filename[MAX_FILE_NAME_LEN];  // 普通文件或目录名称
21     uint32_t i_no;                          // 普通文件或目录对应的 inode 编号
22     enum file_types f_type;            // 文件类型
23 };
…略
```

文件名要存储在目录项中，目录项大小是固定的，因此文件名的长度肯定要有个上限，代码开头定义的宏 MAX_FILE_NAME_LEN 便是文件名的最大长度，其值为 16。其实此宏是为目录项 dir_entry 准备的，下面会说到。

struct dir 是目录结构，它并不在磁盘上存在，只用于与目录相关的操作时，在内存中创建的结构，用

过之后就释放了，不会回写到磁盘中。

其成员 inode 是指针，因此肯定是用于指向内存中 inode，该 inode 必然是在“已打开的 inode 队列”，用到时再说，现在一两句话说不清楚。

成员 dir_pos 用于遍历目录时记录“游标”在目录中的偏移，也就是目录项的偏移量，所以 dir_pos 大小应为目录项大小的整数倍，这与遍历目录的操作相关，用到时再说。

成员 dir_buf 用于目录的数据缓存，如读取目录时，用来存储返回的目录项，这是后话了。

下面是目录项结构 struct dir_entry，它是连接文件名与 inode 的纽带，成员 filename 是文件名，这里只支持最大 16 个字符的文件名。成员 i_no 是文件 filename 对应的 inode 编号，无论 filename 是普通文件，还是目录文件。成员 f_type 是指 filename 的类型，具体类型定义在 fs/fs.h 中，如代码 14-4 所示。

代码 14-4 （project/c14/a/fs/fs.h）

```
…略
 6 #define MAX_FILES_PER_PART 4096
// 每个分区所支持最大创建的文件数
 7 #define BITS_PER_SECTOR 4096      // 每扇区的位数
 8 #define SECTOR_SIZE 512           // 扇区字节大小
 9 #define BLOCK_SIZE SECTOR_SIZE    // 块字节大小
10
11 /* 文件类型 */
12 enum file_types {
13    FT_UNKNOWN,    // 不支持的文件类型
14    FT_REGULAR,    // 普通文件
15    FT_DIRECTORY   // 目录
16 };
```

代码开头定义了一些常用的宏，大伙儿自己看下就行了，下面的枚举结构 file_types 是文件类型，FT_UNKNOWN 的值为 0，表示未知的文件类型，FT_REGULAR 值为 1，表示普通文件，FT_DIRECTORY 值为 2，表示目录。

好啦，本节到这就结束了，下节咱们开始实践。

14.2.2 创建文件系统

本节开始创建文件系统，也就是平时咱们所说的高级格式化分区，相关的代码在 fs/fs.c 中，好啦，不再啰嗦了，开始动真格的。完成格式化分区的函数是 partition_format，它有点长，分成几部分来看，下面先看第一部分，见代码 14-5-1。

代码 14-5-1 （project/c14/a/fs/fs.c）

```
…略
14 /* 格式化分区，也就是初始化分区的元信息，创建文件系统 */
15 static void partition_format(struct disk* hd, struct partition* part) {
16 /* blocks_bitmap_init（为方便实现，一个块大小是一扇区）*/
17     uint32_t boot_sector_sects = 1;
18     uint32_t super_block_sects = 1;
19     uint32_t inode_bitmap_sects =
       DIV_ROUND_UP(MAX_FILES_PER_PART, BITS_PER_SECTOR);
       // I 结点位图占用的扇区数，最多支持 4096 个文件
20     uint32_t inode_table_sects = DIV_ROUND_UP((\
       (sizeof(struct inode) *MAX_FILES_PER_PART)), SECTOR_SIZE);
21     uint32_t used_sects = boot_sector_sects + super_block_sects +\
       inode_bitmap_sects + inode_table_sects;
22     uint32_t free_sects = part->sec_cnt - used_sects;
23
24 /************** 简单处理块位图占据的扇区数 ***************/
25     uint32_t block_bitmap_sects;
26     block_bitmap_sects = DIV_ROUND_UP(free_sects, BITS_PER_SECTOR);
27     /* block_bitmap_bit_len 是位图中位的长度，也是可用块的数量 */
28     uint32_t block_bitmap_bit_len = free_sects - block_bitmap_sects;
29     block_bitmap_sects = DIV_ROUND_UP(block_bitmap_bit_len,\
       BITS_PER_SECTOR);
30 /*******************************************************/
31
```

```
32      /* 超级块初始化 */
33      struct super_block sb;
34      sb.magic = 0x19590318;
35      sb.sec_cnt = part->sec_cnt;
36      sb.inode_cnt = MAX_FILES_PER_PART;
37      sb.part_lba_base = part->start_lba;
38
39      sb.block_bitmap_lba = sb.part_lba_base + 2;
        // 第 0 块是引导块，第 1 块是超级块
40      sb.block_bitmap_sects = block_bitmap_sects;
41
42      sb.inode_bitmap_lba = sb.block_bitmap_lba + sb.block_bitmap_sects;
43      sb.inode_bitmap_sects = inode_bitmap_sects;
44
45      sb.inode_table_lba = sb.inode_bitmap_lba + sb.inode_bitmap_sects;
46      sb.inode_table_sects = inode_table_sects;
47
48      sb.data_start_lba = sb.inode_table_lba + sb.inode_table_sects;
49      sb.root_inode_no = 0;
50      sb.dir_entry_size = sizeof(struct dir_entry);
51
52      printk("%s info:\n", part->name);
53  printk("   magic:0x%x\n   part_lba_base:0x%x\n
all_sectors:0x%x\n   inode_cnt:0x%x\nblock_bitmap_lba:0x%x\n
block_bitmap_sectors:0x%x\n   inode_bitmap_lba:0x%x\n
   inode_bitmap_sectors:0x%x\ninode_table_lba:0x%x\n
   inode_table_sectors:0x%x\ndata_start_lba:0x%x\n",
 sb.magic, sb.part_lba_base, sb.sec_cnt, sb.inode_cnt,
sb.block_bitmap_lba, sb.block_bitmap_sects, sb.inode_bitmap_lba,
sb.inode_bitmap_sects, sb.inode_table_lba,
sb.inode_table_sects, sb.data_start_lba);
```

函数 partition_format 接受 1 个参数，待创建文件系统的分区 part。为方便实现，一个块大小是一扇区，但是相关术语中都是以块单位，因此下面说到“块”时请大家直接理解为扇区。

创建文件系统就是创建文件系统所需要的元信息，这包括超级块位置及大小、空闲块位图的位置及大小、inode 位图的位置及大小、inode 数组的位置及大小、空闲块起始地址、根目录起始地址。创建步骤如下：

（1）根据分区 part 大小，计算分区文件系统各元信息需要的扇区数及位置。

（2）在内存中创建超级块，将以上步骤计算的元信息写入超级块。

（3）将超级块写入磁盘。

（4）将元信息写入磁盘上各自的位置。

（5）将根目录写入磁盘。

下面的创建工作将按照这 5 个步骤依次完成。

代码第 17～18 行是为引导块和超级块占用的扇区数赋值，简单起见，它们均占用 1 扇区大小。顺便提一句，咱们的引导块未使用，因此无所谓大小，但依然要保留其占位。

inode_bitmap_sects 表示 inode 位图占用的扇区数，MAX_FILES_PER_PART 定义在 fs.h 中，表示分区可创建的最大文件数，也就是 inode 数量，它的值为 4096。BITS_PER_SECTOR 同样定义在 fs.h 中，其值也为 4096，经过宏 DIV_ROUND_UP 计算后 inode_bitmap_sects 的值为 1，inode 位图占用 1 扇区。

inode_table_sects 表示 inode 数组占用的扇区数，这是由 inode 的尺寸和数量决定的。

目前已占用的磁盘空间包括引导块、超级块、inode 位图及 inode 数组，现在还差空闲块位图大小未计算出来，它是由空闲块的数量决定的，因此在第 22 行先算出空闲块数量，很简单，空闲块（扇区）数量等于分区总扇区数减去使用的扇区数。

第 25～29 行开始计算空闲块位图占用的扇区数。由于空闲块位图大小和空闲块数量相互依赖，其总和为空闲块数 free_sects，相当于两个互相决定对方大小的变量，咱们这里的处理很简单，先在第 26 行用空闲块数 free_sects 除以每扇区的位数，这样便得到了空闲块位图 block_bitmap 占用的扇区数 block_bitmap_sects。空闲块位图占用了一部分空闲扇区，因此现在真正的空闲块数得把 block_bitmap_sects 从 free_sets 中减去，其结果也是位图中位的个数，即在第 28 行我们把结果写入了变量 block_bitmap_bit_len，然后再用变量

block_bitmap_bit_len 重新除以 BITS_PER_SECTOR，这便是空闲块位图最终占用的扇区数 block_bitmap_sects。虽然这样处理显得粗暴一些，但确实很简单。

接下来第 33 行开始在内存中创建超级块，此处用局部变量生成超级块，用的是栈中的内存，不过还好，超级块是 512 字节，栈还够用。一直到第 48 行都没什么好说的，比较清楚，第 49 行的代码"sb.root_inode_no = 0"，这表示根目录的 inode 编号为 0，也就是说 inode 数组中第 0 个 inode 我们留给了根目录。

第 50 行为目录项尺寸 dir_entry_size 赋值。第 52～53 行打印超级块中元信息，我们在运行时会看到它们。

好啦，下面看第二部分代码，见代码 14-5-2。

代码 14-5-2　（project/c14/a/fs/fs.c）

```
…略
55      struct disk* hd = part->my_disk;
56 /********************************
57  * 1 将超级块写入本分区的 1 扇区 *
58  ******************************/
59     ide_write(hd, part->start_lba + 1, &sb, 1);
60     printk("   super_block_lba:0x%x\n", part->start_lba + 1);
61
62 /* 找出数据量最大的元信息，用其尺寸做存储缓冲区*/
63     uint32_t buf_size = (sb.block_bitmap_sects >= \
       sb.inode_bitmap_sects ? sb.block_bitmap_sects : sb.inode_bitmap_sects);

64     buf_size = (buf_size >= sb.inode_table_sects ? \
buf_size : sb.inode_table_sects) * SECTOR_SIZE;

65    uint8_t* buf = (uint8_t*)sys_malloc(buf_size);
// 申请的内存由内存管理系统清 0 后返回
66
67 /***************************************
68  * 2 将块位图初始化并写入 sb.block_bitmap_lba *
69  *************************************/
70     /* 初始化块位图 block_bitmap */
71     buf[0] |= 0x01;       // 第 0 个块预留给根目录，位图中先占位
72     uint32_t block_bitmap_last_byte = block_bitmap_bit_len / 8;
73     uint8_t  block_bitmap_last_bit  = block_bitmap_bit_len % 8;
74     uint32_t last_size = SECTOR_SIZE - \
                         (block_bitmap_last_byte % SECTOR_SIZE);
       // last_size 是位图所在最后一个扇区中，不足一扇区的其余部分
75
76     /* 1 先将位图最后一字节到其所在的扇区的结束全置为 1，
       即超出实际块数的部分直接置为已占用*/
77     memset(&buf[block_bitmap_last_byte], 0xff, last_size);
78
79     /* 2 再将上一步中覆盖的最后一字节内的有效位重新置 0 */
80     uint8_t bit_idx = 0;
81     while (bit_idx <= block_bitmap_last_bit) {
82         buf[block_bitmap_last_byte] &= ~(1 << bit_idx++);
83     }
84     ide_write(hd, sb.block_bitmap_lba, buf, sb.block_bitmap_sects);
85
```

第 55 行获取分区 part 自己所属的硬盘 hd，hd 将作为后续参数。

超级块已经构建完成，在第 59 行将其写到本分区开始的扇区加 1 的地方，即 part->start_lba + 1，也就是跨过引导扇区，把超级块写入引导扇区后面的扇区中。尽管此处的引导块没什么用，但也要将其位置空出来。

元信息最终是要写入硬盘的，但数据源是在内存中。像超级块本身是 1 扇区大小，我们是用局部变量声明它的，栈还能将就对付。可是空闲块位图、inode 数组位图等占用的扇区数较大（剧透：好几百扇区），所以这里不便用局部变量来保存它们了，应该从堆中申请内存获取缓冲区，最好是找出数据量最大的元信息，用其尺寸作为申请的内存大小。在第 63～64 行开始选出占用空间最大的元信息，使其尺寸作为申请的缓冲区大小，在第 65 行申请内存返回给指针 buf，buf 是通用的缓冲区，接下来往磁盘中的数据写入操作都将 buf 作为数据源，通过不同的类型转换，使 buf 变成合适的缓冲区类型。

接下来是把块位图写入磁盘扇区 sb.block_bitmap_lba 的准备工作。

我们把第 0 个空闲块作为根目录，因此我们需要在空闲块位图中将第 0 位置 1，这是通过第 71 行代码完成的。

接下来第 72～83 行把块位图最后一个扇区中不属于空闲块的位初始为 1。这么做的原因是这样的，可得好好说道说道。

文件系统的主要工作是资源管理，跟踪资源的状态是通过位图来实现的，因此创建文件系统就是创建各种资源的位图，位图肯定是在内存中先创建好，然后再将位图持久化到硬盘，“持久化”是指把数据写到可以长久保存信息的存储介质上，永远保存，比如磁盘就是一种持久化存储介质。虽然我们此时是在创建位图，但位图并不是在本次使用，而是将来挂载本分区的时候再用。为什么呢？您想，分区挂载的目的是为了使用分区上的数据，这免不了数据资源的增删改查，为了跟踪分区中的数据资源使用情况，肯定还是需要这些资源的位图。而位图的操作肯定是在内存中，但挂载分区前内存中可没有位图，这就需要事先把这些位图提前持久化到磁盘上（这正是咱们目前所做的工作），挂载分区时再把位图从硬盘上加载到内存。由于内存不能长久保存数据，断电之后内存中的一切都会灰飞烟灭，所以即使是将来挂载分区后，内存中的位图哪怕只是更改了一个位，都要及时同步持久化到磁盘上。但是您看到了，现在咱们创建的块位图大小是通过宏 DIV_ROUND_UP 把 free_sects 除以 BITS_PER_SECTOR 向上取整得到的，因此位图中最后 1 扇区的末尾在大多数情况下都会有多余的位，也就是说这些多余的位并不代表某个资源，或者说是这些位指代的“真正资源”并不存在，而是指向了硬盘上“其他不属于位图所指代资源的数据”，若在位图操作中强行使用这些位的话，必然会损坏那些资源外的数据。我们现在持久化到磁盘上的位图是以扇区为单位的，将来挂载分区时，再从磁盘上恢复的位图也会包括扇区末尾那些无效、多余的位，因此现在必须将这些多余位初始为 1，将来就不会再分配这些位对应的资源了，避免了出现错误的情况。也许有同学会问，位图中不是有 btmp_bytes_len 限制吗，访问的位肯定不能超过 btmp_bytes_len 的范围。是的，可是 btmp_bytes_len 的值哪里来呢？难道我们在创建文件系统时将资源的数量（也就是 btmp_bytes_len）写到超级块或某个扇区中吗？确实是可以这么做，但还是觉得有些不雅，毕竟磁盘上的位图是唯一的位图来源，它反应了最真实的资源情况，资源的状态应该以它为准。我们完全可以在加载位图后再计算出 btmp_bytes_len 的值，而且避免了二次读硬盘的操作。剧透一下，将来挂载分区时把位图从硬盘加载到内存后，内存中位图的 btmp_bytes_len=位图占用的扇区数*每扇区的位数。因此在将位图持久化到硬盘之前，一定要将位图最后一扇区中的多余位初始为 1，表示它们已被占用，不许再碰啦。道理讲完了，代码部分见注释吧，其实很简单。

待块位图初始化完成后，在第 84 行将其写入硬盘 sb.block_bitmap_lba 处。

好啦，下面看第三部分代码，见代码 14-5-3。

代码 14-5-3　（project/c14/a/fs/fs.c）

```
…略
 86 /****************************************
 87   * 3 将 inode 位图初始化并写入 sb.inode_bitmap_lba *
 88   ***************************************/
 89    /* 先清空缓冲区*/
 90    memset(buf, 0, buf_size);
 91    buf[0] |= 0x1;      // 第 0 个 inode 分给了根目录
 92    /* 由于 inode_table 中共 4096 个 inode,
        * 位图 inode_bitmap 正好占用 1 扇区,
 93     * 即 inode_bitmap_sects 等于 1,
        * 所以位图中的位全都代表 inode_table 中的 inode,
 94     * 无需再像 block_bitmap 那样单独处理最后一扇区的剩余部分,
 95     * inode_bitmap 所在的扇区中没有多余的无效位 */
 96    ide_write(hd, sb.inode_bitmap_lba, buf, sb.inode_bitmap_sects);
 97
 98 /****************************************
 99   * 4 将 inode 数组初始化并写入 sb.inode_table_lba *
100   ***************************************/
101   /* 准备写 inode_table 中的第 0 项,即根目录所在的 inode */
102     memset(buf, 0, buf_size);  // 先清空缓冲区 buf
103     struct inode* i = (struct inode*)buf;
104     i->i_size = sb.dir_entry_size * 2;    // .和..
105     i->i_no = 0;   // 根目录占 inode 数组中第 0 个 inode
```

```
106     i->i_sectors[0] = sb.data_start_lba;
// 由于上面的memset，i_sectors数组的其他元素都初始化为0

107     ide_write(hd, sb.inode_table_lba, buf, sb.inode_table_sects);
108
109 /***************************************
110  * 5 将根目录写入sb.data_start_lba
111  ***************************************/
112    /* 写入根目录的两个目录项.和.. */
113    memset(buf, 0, buf_size);
114    struct dir_entry* p_de = (struct dir_entry*)buf;
115
116    /* 初始化当前目录"." */
117    memcpy(p_de->filename, ".", 1);
118    p_de->i_no = 0;
119    p_de->f_type = FT_DIRECTORY;
120    p_de++;
121
122    /* 初始化当前目录父目录".." */
123    memcpy(p_de->filename, "..", 2);
124    p_de->i_no = 0;   // 根目录的父目录依然是根目录自己
125    p_de->f_type = FT_DIRECTORY;
126
127    /* sb.data_start_lba已经分配给了根目录，里面是根目录的目录项 */
128    ide_write(hd, sb.data_start_lba, buf, 1);
129
130    printk("   root_dir_lba:0x%x\n", sb.data_start_lba);
131    printk("%s format done\n", part->name);
132    sys_free(buf);
133 }
```

代码第86行开始准备将inode位图写入磁盘扇区sb.inode_bitmap_lba，第91行将inode位图（buf）中第0个inode置为1，原因是我们把inode数组中第0个inode分配给了根目录。

分区只支持4096个文件，也就是inode数组inode_table中共4096个inode，inode位图inode_bitmap正好占用1扇区，即inode_bitmap_sects等于1，inode_bitmap所在的扇区中没有多余的无效位，因此无需再像块位图block_bitmap那样单独处理最后一扇区的剩余部分。最后在第96行将inode位图写入磁盘。

接下来准备把inode数组写入sb.inode_table_lba。

inode_table_sects是通过宏DIV_ROUND_UP除法向上取整得到的结果，因此inode_table最终在磁盘上占据的全部扇区中，并不是所有空间都是inode_table的内容，大多数情况下在其所占据的最后不足一扇区，剩余部分肯定不属于inode_table中的inode。不过由于inode数量是由inode_bitmap来控制的，只要保证inode_bitmap不越界就行，而inode_bitmap占用完整1扇区，里面没有多余的位，因此肯定是正确的，不用再额外处理什么。好啦，下面介绍实际代码。我们把第0个inode已经分配给了根目录，因此现在要初始化第0个inode为根目录的信息。我们前面讨论过的，当inode指代的是普通文件时，i_size是文件大小，当inode指代的是目录时，成员i_size表示此目录下所有目录项的大小之和。任何目录中默认都会有表示当前目录的“.”和上一级目录“..”，根目录也不能例外，这两个目录项的大小之和便是此inode中i_size的值，因此先在第103行将buf转换为inode结构struct inode型指针后，通过第104行的代码“i->i_size = sb.dir_entry_ size * 2”将i_size赋值为两个目录项的大小。第105行的代码“i->i_no = 0”为inode编号赋值为0，表示此inode自己是inode数组中第0个inode。第106行的代码“i->i_sectors[0] = sb.data_start_lba”使此inode的第0个数据块指向sb.data_start_lba，也就是我们把根目录安排在最开始的空闲块中。由于第102行的memset清0工作，i_sectors数组的其他元素也都被初始化为0。最后在第107行，将inode数组写入硬盘。

最后一项工作是在根目录中写目录项”.”和”..”。任何目录都有这两个目录项，”.”表示当前目录，”..”表示上一级目录。

第114行将buf转换为目录项struct dir_entry型指针，此时p_de指向buf，接下来先对第1个目录项”.”初始化。在第117行通过memcpy函数把”.”写入目录项的filename成员，接下来的2行分别为目录项的i_no赋值为0，使其指向根目录自己，为目录项的f_type赋值为FT_DIRECTORY，使其类型为目录。第120行的p_de++执行过后，p_de指向下一目录项”..”。

第 123～125 行所做的工作是初始化父目录“..”，原理同当前目录“.”一样，不再赘述。接下来在第 128 行将根目录写入磁盘。最后第 132 行释放缓冲区 buf。

有关创建文件系统的代码到这就结束了，现在还没有调用它的方法，接下来仍然在 fs.c 中添加个函数 filesys_init，它是文件系统初始化函数，如代码 14-5-4 所示。

代码 14-5-4　（project/c14/a/fs/fs.c）

```
…略
135 /* 在磁盘上搜索文件系统，若没有则格式化分区创建文件系统 */
136 void filesys_init() {
137    uint8_t channel_no = 0, dev_no, part_idx = 0;
138
139    /* sb_buf 用来存储从硬盘上读入的超级块 */
140    struct super_block* sb_buf = \
       (struct super_block*)sys_malloc(SECTOR_SIZE);
141
142    if (sb_buf == NULL) {
143       PANIC("alloc memory failed!");
144    }
145    printk("searching filesystem......\n");
146    while (channel_no < channel_cnt) {
147       dev_no = 0;
148       while(dev_no < 2) {
149          if (dev_no == 0) {   // 跨过裸盘 hd60M.img
150             dev_no++;
151             continue;
152         }
153         struct disk* hd = &channels[channel_no].devices[dev_no];
154         struct partition* part = hd->prim_parts;
155         while(part_idx < 12) {   // 4 个主分区+8 个逻辑
156            if (part_idx == 4) {   // 开始处理逻辑分区
157               part = hd->logic_parts;
158            }
159
160    /* channels 数组是全局变量，默认值为 0，disk 属于其嵌套结构，
161     * partition 又为 disk 的嵌套结构，因此 partition 中的成员默认也为 0。
162     * 若 partition 未初始化，则 partition 中的成员仍为 0。
163     * 下面处理存在的分区 */
164            if (part->sec_cnt != 0) {  // 如果分区存在
165               memset(sb_buf, 0, SECTOR_SIZE);
166
167    /* 读出分区的超级块，根据魔数是否正确来判断是否存在文件系统 */
168               ide_read(hd, part->start_lba + 1, sb_buf, 1);
169
170    /* 只支持自己的文件系统，若磁盘上已经有文件系统就不再格式化了 */
171               if (sb_buf->magic == 0x19590318) {
172                  printk("%s has filesystem\n", part->name);
173               } else { // 其他文件系统不支持，一律按无文件系统处理
174                  printk("formatting %s`s partition %s......\n",\
                     hd->name, part->name);
175                  partition_format(part);
176               }
177            }
178            part_idx++;
179            part++;    // 下一分区
180         }
181         dev_no++;     // 下一磁盘
182      }
183      channel_no++;    // 下一通道
184    }
185    sys_free(sb_buf);
186 }
```

代码比较容易，咱们就不说那么细了，从第 146 行开始在分区上扫描文件系统，我们这里只支持 partition_format 创建的文件系统，其魔数等于 0x19590318，如果未发现魔数为 0x19590318 的文件系统就调用 partition_format 去创建。这里面是通过三层循环完成的，最外层循环用来遍历通道，中间层循环用来遍历通道中的硬盘，最内层循环用来遍历硬盘上的所有分区。

咱们对每个硬盘最多支持 12 个分区，即 4 个主分区和 8 个逻辑分区，因此在第 155 行遍历硬盘分区时只需要循环 12 次。part 用于指向每一个分区，在第 156 行，当分区索引变量 part_idx 等于 4 时，这表示全部主分区都处理完了，于是在第 157 行将 part 指向硬盘的逻辑分区数组，也就是第一个逻辑分区的地址。

虽然我们支持 12 个分区，但并不表示硬盘中存在 12 个分区，因此在进行格式化分区之前，先在第 164 行判断分区是否存在，这是通过分区的 sec_cnt 是否为 0 来判断的，之所以可以用此变量来判断，原因是分区 part 所在的硬盘作为全局数组 channels 的内嵌成员，全局变量会被初始化为 0。我们在扫描分区表的时候会把分区的信息写到 part 中，因此只要分区不存在，分区 part 中任意成员的值都会是 0，只是我们这里用 sec_cnt 来判断而已。

第 167 行开始读分区的超级块，目的是获取超级块中的魔数。在第 171 行若判断出魔数为 0x19590318，就表示已经有文件系统，不再去格式它。否则在第 175 行对此分区执行格式化。其他部分不多说了。

好啦，本节的代码就这些，马上编译运行，看看效果，如图 14-12 所示。

```
Bochs x86 emulator, http://bochs.sourceforge.net/
   inode_bitmap_lba:0x162A9
   inode_bitmap_sectors:0x1
   inode_table_lba:0x162AA
   inode_table_sectors:0x260
   data_start_lba:0x1650A
   super_block_lba:0x162A0
   root_dir_lba:0x1650A
sdb8 format done
formatting sdb`s partition sdb9......
sdb9 info:
   magic:0x19590318
   part_lba_base:0x1D8BF
   all_sectors:0xA521
   inode_cnt:0x1000
   block_bitmap_lba:0x1D8C1
   block_bitmap_sectors:0xB
   inode_bitmap_lba:0x1D8CC
   inode_bitmap_sectors:0x1
   inode_table_lba:0x1D8CD
   inode_table_sectors:0x260
   data_start_lba:0x1DB2D
   super_block_lba:0x1D8C0
   root_dir_lba:0x1DB2D
sdb9 format done
IPS: 28.946M
```

▲图 14-12　分区格式化

硬盘 hd80M.img 中分区较多，包括 1 个主分区、5 个逻辑分区，因此输出信息较多，图 14-12 所示是最后两个分区的格式化信息。当下一次运行时，由于已经有了文件系统，因此不会再进行格式化工作，运行结果如图 14-13 所示。

```
Bochs x86 emulator, http://bochs.sourceforge.net/
      MODULE: Generic 1234
      SECTORS: 121968
      CAPACITY: 59MB
   disk sdb info:
      SN: BXHD00012
      MODULE: Generic 1234
      SECTORS: 163296
      CAPACITY: 79MB

   all partition info
   sdb1 start_lba:0x3F, sec_cnt:0x7DC1
   sdb5 start_lba:0x7E3F, sec_cnt:0x46A1
   sdb6 start_lba:0xC51F, sec_cnt:0x6231
   sdb7 start_lba:0x1278F, sec_cnt:0x3AD1
   sdb8 start_lba:0x1629F, sec_cnt:0x75E1
   sdb9 start_lba:0x1D8BF, sec_cnt:0xA521
ide_init done
searching filesystem......
sdb1 has filesystem
sdb5 has filesystem
sdb6 has filesystem
sdb7 has filesystem
sdb8 has filesystem
sdb9 has filesystem
CTRL + 3rd button enables mouse
```

▲图 14-13　检测到文件系统

本节有些长，不过总算结束了，大伙辛苦了，咱们下节继续。

14.2.3　挂载分区

如果大伙儿最先使用的操作系统是 Windows 的话，想必您还记得刚接触过 Linux 的时候，一定对其目录

结构很不适应。Windows 系统的分区盘符简单明了，C、D、E 盘直接摆在那，用不用都看得到。而 Linux 中却大不相同，分区使用的时候可以单独“拿”出来，不用的时候还可以“收”起来。

Linux 内核所在的分区是默认分区，自系统启动后就以该分区为默认分区，该分区的根目录是固定存在的，要想使用其他新分区的话，需要用 mount 命令手动把新的分区挂载到默认分区的某个目录下，这就是上面所说的“拿”出来。尽管其他分区都有自己的根目录，但是默认分区的根目录才是所有分区的父目录，因此挂载分区之后，整个路径树就像一串葡萄。分区不用的时候还可以通过 umount 命令卸载，这就是上面所说的“收”起来。

我们要想实现文件操作，肯定要指定操作哪个分区，对哪个分区上的文件执行读写操作。磁盘上的分区很多，我们需要随意操作任何一个分区，因此我们要实现分区挂载的功能。此功能类似 Linux 的 mount 命令的功能，但又不完全是，因为 mount 命令是把一个分区挂载到默认分区（操作系统所在分区）的某个目录上，但咱们的操作系统安装到裸盘 hd60M.img 上，上面没有分区，更谈不上文件系统了。

其实要想把操作系统安装到文件系统上，必须在实现内核之前先实现文件系统模块，至少得完成写文件的功能，然后把操作系统通过写文件功能写到文件系统上。举个例子，大伙儿还对安装操作系统有印象吧，无论是安装 Windows，还是 Linux，安装过程中都是先选择安装到哪个分区上，然后选择以什么文件系统来格式化该分区，Windows 系统通常是 fat32 或 ntfs，Linux 系统通常是 ext2 或 ext3，在为该分区格式化出文件系统之后才开始正式的安装，安装界面通常是介绍系统的最新特性之类，最终操作系统就被安装到某种文件系统上了。尽管我们也可以这么做，但这样做的话我怕会在本书的开头就吓跑很多兄弟，一开始就很麻烦的话确实会打击学习兴趣，毕竟咱们是在学习操作系统的实现原理，而不是真的像商业操作系统那样做得有模有样，而且不怕您笑话，小弟我确实能力有限，也不曾想过先创建文件系统再创建内核，哈哈。好啦，回来说正经的，既然操作系统不在文件系统上，也就没有默认目录，那就不用考虑分区挂载到哪个目录下，因此咱们要实现的挂载功能很简单——直接选择待操作的分区。

挂载分区的实质是把该分区文件系统的元信息从硬盘上读出来加载到内存中，这样硬盘资源的变化都用内存中元信息来跟踪，如果有写操作，及时将内存中的元信息同步写入到硬盘以持久化。还记得上节咱们对持久化的讨论吗？就是为了现在做准备，不多说了，上代码，见代码 14-6。

代码 14-6　（project/c14/b/fs/fs.c）

```
…略
 14 struct partition* cur_part;     // 默认情况下操作的是哪个分区
 15
 16 /* 在分区链表中找到名为 part_name 的分区，并将其指针赋值给 cur_part */
 17 static bool mount_partition(struct list_elem* pelem, int arg) {
 18     char* part_name = (char*)arg;
 19     struct partition* part = elem2entry(struct partition, part_tag, pelem);
 20     if (!strcmp(part->name, part_name)) {
 21         cur_part = part;
 22         struct disk* hd = cur_part->my_disk;
 23
 24         /* sb_buf 用来存储从硬盘上读入的超级块 */
 25         struct super_block* sb_buf = \
           (struct super_block*)sys_malloc(SECTOR_SIZE);
 26
 27         /* 在内存中创建分区 cur_part 的超级块 */
 28         cur_part->sb = (struct super_block*)\
           sys_malloc(sizeof(struct super_block));
 29         if (cur_part->sb == NULL) {
 30             PANIC("alloc memory failed!");
 31         }
 32
 33         /* 读入超级块 */
 34         memset(sb_buf, 0, SECTOR_SIZE);
 35         ide_read(hd, cur_part->start_lba + 1, sb_buf, 1);
 36
 37         /* 把 sb_buf 中超级块的信息复制到分区的超级块 sb 中 */
 38         memcpy(cur_part->sb, sb_buf, sizeof(struct super_block));
 39
 40   /**********     将硬盘上的块位图读入到内存     **************/
```

```
41        cur_part->block_bitmap.bits =\
         (uint8_t*)sys_malloc(sb_buf->block_bitmap_sects * SECTOR_SIZE);

42        if (cur_part->block_bitmap.bits == NULL) {
43            PANIC("alloc memory failed!");
44        }
45        cur_part->block_bitmap.btmp_bytes_len =\
          sb_buf->block_bitmap_sects * SECTOR_SIZE;

46        /* 从硬盘上读入块位图到分区的 block_bitmap.bits */
47        ide_read(hd, sb_buf->block_bitmap_lba, \
          cur_part->block_bitmap.bits, sb_buf->block_bitmap_sects);
48        /*************************************************************/
49
50  /**********     将硬盘上的 inode 位图读入到内存    ************/
51        cur_part->inode_bitmap.bits =\
         (uint8_t*)sys_malloc(sb_buf->inode_bitmap_sects * SECTOR_SIZE);

52        if (cur_part->inode_bitmap.bits == NULL) {
53           PANIC("alloc memory failed!");
54        }
55        cur_part->inode_bitmap.btmp_bytes_len =\
          sb_buf->inode_bitmap_sects * SECTOR_SIZE;

56        /* 从硬盘上读入 inode 位图到分区的 inode_bitmap.bits */
57        ide_read(hd, sb_buf->inode_bitmap_lba, \
          cur_part->inode_bitmap.bits, sb_buf->inode_bitmap_sects);
58        /*************************************************************/
59
60        list_init(&cur_part->open_inodes);
61        printk("mount %s done!\n", part->name);
62
63 /* 此处返回 true 是为了迎合主调函数 list_traversal 的实现，
    * 与函数本身功能无关。
64  * 只有返回 true 时 list_traversal 才会停止遍历，
    * 减少了后面元素无意义的遍历 */
65        return true;
66     }
67     return false;    // 使 list_traversal 继续遍历
68 }
…略
192 void filesys_init() {
…略
243    /* 确定默认操作的分区 */
244    char default_part[8] = "sdb1";
245    /* 挂载分区 */
246    list_traversal(&partition_list, mount_partition, (int)default_part);
247 }
```

我们先看代码第 244 行，default_part 为默认分区的名称，其值为 sdb1，也就是说我们默认操作的分区是 sdb1。分区挂载是借助函数 list_traversal 完成的，第 246 行的“list_traversal(&partition_list, mount_partition, (int)default_part);”，功能相当于用 mount_partition(default_part)处理每一个分区。其中 partition_list 是所有分区的列表，mount_partition 是一会我们要介绍的挂载分区的函数，(int)default_part 将数组地址转换成整型作为 mount_partition 的参数，参数转换为整型的原因是 list_traversal 原型是“struct list_elem* list_traversal(struct list* plist, function func, int arg)”，其功能是遍历 plist 中所有元素，直到 func(arg)返回 true 或者列表元素全部遍历结束。

好啦，我们介绍下 mount_partition，请移步代码第 17 行。

函数 mount_partition 是 list_traversal 的回调函数，其接受两个参数，pelem 是 list_traversal 传给它的列表中的元素，在此处 pelem 是分区 partition 中的 part_tag 的地址。arg 是待比对的参数，此处是分区名。函数功能是在分区链表中找到免为 part_name 的分区，并将其指针赋值给 cur_part。cur_part 是在第 14 行定义的全局变量，它用来记录默认操作的分区。

第 18 行将 arg 还原为字符指针，赋值给 part_name。第 19 行将 pelem 通过宏 elem2entry 还原为分区 part。

第 20 行，通过 strcmp 比对 part->name 和 part_name，如果相等则找到了该分区，接下来开始加载该

分区的元信息。

首先在第 21 行将找到的分区指针 part 赋值给 cur_part，也就是说找到了默认的分区。然后在第 22 行获得分区所在的硬盘 hd，hd 将作为后续硬盘操作的参数。

第 25 行申请了一扇区大小的内存作为缓冲区 sb_buf，sb_buf 用于容纳从硬盘读取的 1 扇区大小的超级块。

第 28 行创建分区 cur_part 的超级块 cur_part->sb，为此在第 28 行申请超级块大小的内存，然后赋值给 cur_part->sb。内存申请成功后，在第 35 行读入超级块到 sb_buf，然后在第 38 行将缓冲区 sb_buf 中超级块的信息复制到 cur_part->sb，这么做的原因是超级块虽然占一扇区，但我们的 struct super_block 结构并没有一扇区大小，所以只把超级块中有用的信息复制过来，那些用于填充的超级块的数组“uint8_t pad[460]”就不要了，毕竟没用，还占内存。

第 41 行为当前分区 cur_part 的块位图申请内存，内存大小是超级块中的 block_bitmap_sects 乘以 SECTOR_SIZE。

第 45 行初始化块位图的 btmp_bytes_len，上节讨论持久化元信息的时候剧透过，用块位图占用的扇区数 block_bitmap_sects 乘以扇区大小 SECTOR_SIZE 的值作为位图 btmp_bytes_len 的值。

第 47 行将硬盘上的块位图读入到内存，也就是读到 cur_part->block_bitmap.bits，完成了块位图的加载。分区越大，块位图占用的内存就越多，因此物理内存一定要匹配，不要出现物理内存太小而分区太大，连块位图都容纳不了的情况，哈哈，我这里说的是咱们虚拟机的配置，线上服务器都不会。

第 50～58 行是将 inode 位图读入到内存，与载入块位图同理，不说了。

第 60 行初始化分区的 open_inodes 列表。到此函数 mount_partition 介绍完了。

编译运行，结果如图 14-14 所示。

```
Bochs x86 emulator, http://bochs.sourceforge.net/
      SECTORS: 121968
      CAPACITY: 59MB
   disk sdb info:
      SN: BXHD00012
      MODULE: Generic 1234
      SECTORS: 163296
      CAPACITY: 79MB

   all partition info
   sdb1 start_lba:0x3F, sec_cnt:0x7DC1
   sdb5 start_lba:0x7E3F, sec_cnt:0x46A1
   sdb6 start_lba:0xC51F, sec_cnt:0x6231
   sdb7 start_lba:0x1278F, sec_cnt:0x3AD1
   sdb8 start_lba:0x1629F, sec_cnt:0x75E1
   sdb9 start_lba:0x1D8BF, sec_cnt:0xA521
ide_init done
searching filesystem......
sdb1 has filesystem
sdb5 has filesystem
sdb6 has filesystem
sdb7 has filesystem
sdb8 has filesystem
sdb9 has filesystem
mount sdb1 done!
IPS: 29.560M
```

▲图 14-14　分区挂载演示

哈哈，图上并没有什么实质的信息，就在最下面打印出“mount sdb1 done!”，挂载显得好“低调”。好啦，本节就到这，下节再见啦。

14.3 文件描述符简介

14.3.1　文件描述符原理

Linux 中所有文件操作都基于文件描述符，为此咱们简短介绍一下这个概念。

在这之前，我们已经知道文件是用 inode 来表示的，个人觉得，inode 其实也可称为“文件数据块描述符”（纯属个人杜撰，仅是为了突显与文件描述符的区别），用于描述文件的存储、权限等。但 inode 是操作系统为自己的文件系统准备的数据结构，它用于文件存储的管理，与用户关系不大，咱们要介绍的文

件描述符才是与用户息息相关的。

文件描述符即 file descriptor，但凡叫“描述符”的数据结构都用于描述一个对象，文件描述符所描述的对象是文件的操作。为了搞清楚文件描述符的意义，咱们先看下它与 inode 的区别和联系。

还是拿 Linux 举例，读写文件的本质是：先通过文件的 inode 找到文件数据块的扇区地址，随后读写该扇区，从而实现了文件的读写。几乎所有的操作系统都允许一个进程同时、多次、打开同一个文件（并不关闭），同样该文件也可以被多个不同的进程同时打开。为实现文件任意位置的读写，执行读写操作时可以指定偏移量作为该文件内的起始地址，此偏移量相当于文件内的指针。也就是说，该文件每被打开一次，文件读写的偏移量都可以任意指定，即对同一个文件的多次读写都是各自操作各自的，任意一个文件操作的偏移量都不影响其他文件操作的偏移量。注意，这里所说的是“互不影响”是指文件内的“偏移量”，并不是文件内容，因为文件内容是共享的，对文件内容的修改必然会相互影响。另外，通常情况下对文件的操作都只是读写文件的一小部分数据，即使想读写整个文件的话，由于一般文件的体积都较大，而内存缓冲区较小，所以文件的读写并不是一次性从头到尾操作整个文件，往往是通过连续多次的小数据量读写完成的，下一次读写的位置必须以上一次的读写位置为起始，因此，文件系统需要把任意时刻的偏移量记录下来。问题来了，偏移量应该记录在哪里呢？inode 中肯定不记录这些与文件操作相关的数据，人家只把有限的空间记录与存储相关的信息。为解决这个问题，Linux 提供了称为“文件结构”的数据结构（也称为 file 结构），专门用于记录与文件操作相关的信息，每次打开一个文件就会产生一个文件结构，多次打开该文件就为该文件生成多个文件结构，各自文件操作的偏移量分别记录在不同的文件结构中，从而实现了“即使同一个文件被同时多次打开，各自操作的偏移量也互不影响”的灵活性。文件结构的逻辑表示如图 14-15 所示，这是文件结构中最基本的数据成员。

fd_pos	// 文件偏移量
fd_flag	// 文件打开的标志如 O_CREAT
fd_inode	// inode 指针
...	

▲图 14-15　文件描述符逻辑结构

Linux 把所有的“文件结构”组织到一起形成数组统一管理，该数组称为文件表，咳咳，我们要多次引用此概念。

也许现在您感觉有些模糊，似乎无法将相关概念贯穿起来，不急，毕竟我还没介绍完呢，咱们一步一个脚印，先总结一下，inode 用于描述文件存储相关信息，文件结构用于描述“文件打开”后，文件读写偏移量等信息。文件与 inode 一一对应，一个文件仅有一个 inode，一个 inode 仅对应一个文件。一个文件可以被多次打开，因此一个 inode 可以有多个文件结构，多个文件结构可以对应同一个 inode。

现在该说说文件描述符了，在 Linux 中，我们读写函数文件时都是通过操作文件描述符来完成的。举个例子，拿 open 函数来说，其原型为 int open(const char *pathname, int flags)，pathname 是待打开的文件路径及文件名，flag 是打开标识，调用它之后，系统会返回文件 pathname 的文件描述符。不过这些都不重要，重要的是返回值类型为 int，这说明返回值是个整型数字。数字？我怎么记得返回的是文件描述符？没错，这不冲突，open 是返回一个数字，而该数字就是我们所说的文件描述符，文件描述符确实只是个整数，准确地说，它是 PCB 中文件描述符数组元素的下标，只不过此数字并不用来表示“数量”，而是用来表示“位置”，它是位于进程 PCB 中的文件描述符数组的元素的下标，而文件描述符数组元素中的信息又指向文件表中的某个文件结构。对此感到费解？咱们慢慢说。

在 Linux 中每个进程都有单独的、完全相同的一套文件描述符，因此它们与其他进程的文件描述符互不干涉，这些文件描述符被组织成文件描述符数组统一管理。您看到了，图 14-15 是最基本的文件结构，这 3 个成员要占用十几字节的空间，前面说过了，打开一个文件时会产生一个文件结构，这要是该任务同时打开 N 多个文件，文件表可就大了。通常情况下为避免文件表占用过大的内存空间，文件结构的数量必须是有限的，这就是进程可打开的最大文件数有限的原因（在 Linux 中可用 ulimit 命令来修改）。文件描述符数组中的前 3 个都是标准的文件描述符，如文件描述符 0 表示标准输入，1 表示标准输出，2 表示标准错误。为什么文件描述符是数字，而不是像其他描述符那样，是个具有多个成员属性的复合数据结构？原因有两个。

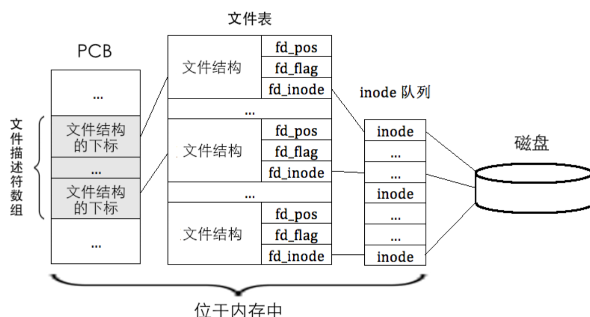
（1）为了一视同仁，使各进程可打开的文件数是一样的，各进程必须有独立的、大小完全一样的一套文件描述符数组，而不能所有进程共享同一套文件描述符数组，比如 A、B 两个进程都可以用文件描述符 9 指向任意文件（相同或不同的文件都可以）。相反，如果所有进程共享同一套文件描述符数组的话，如

果进程 A 占用了文件描述符 9，进程 B 只有使用其他空闲的文件描述符。

（2）文件结构中包含进程执行文件操作的偏移量，它属于与各个任务单独绑定的资源，因此最好放在 PCB 中管理。但当进程打开的文件数增多的时候，文件表（由文件结构组成的数组）占用的空间较大，而 PCB 占用的内存通常就是几个页框，Linux 中的 PCB 也只是 2 页框大小，咱们 PCB 只占用 1 页框。

结合以上两个原因，通常情况下不会把“真正的、庞大的”文件表塞到狭小的 PCB 中，一般只要在 PCB 中建立个文件描述符数组就可以了，该数组成员不需要是真正的文件结构（至少包括 3 个成员），出于简单处理，咱们用 int 整型就足够了，用它存储文件表中文件结构的下标，如果您愿意的话，可以用位图或链表为任务支持更多，甚至无限的最大打开文件数。当用户进程打开文件时，文件系统给用户进程返回的是该进程 PCB 中文件描述符数组下标值，也就是文件描述符。

Linux 中的文件操作通常是先用 open 打开文件以获取该文件的文件描述符，然后将文件描述符作为参数传给 read 或 write 等函数对该文件进行读写操作。大家肯定想知道 Linux 是如何通过一个简单的数字，也就是文件描述符来找到文件数据块的，话说要想了解此过程，必须要了解此过程涉及的数据结构及过程组织形式。咱们就不卖关子了，整个过程全景图如图 14-16 所示。



▲图 14-16　文件描述符与 inode 关联关系

图 14-16 是 Linux 通过文件描述符查找文件数据块的过程，这涉及到以下三个数据结构，它们都是位于内存中的。

（1）PCB 中的文件描述符数组。

（2）存储所有文件结构的文件表。

（3）inode 队列，也就是 inode 缓存。

现在从左到右梳理一下图 14-16，某进程把文件描述符作为参数提交给文件系统时，文件系统用此文件描述符在该进程的 PCB 中的文件描述符数组中索引对应的元素，从该元素中获取对应的文件结构的下标，用该下标在文件表中索引相应的文件结构，从该文件结构中获取文件的 inode，最终找到了文件的数据块。提示一下，若该 inode 在 inode 队列中不存在，此时会多一个处理过程：文件系统会从硬盘上将该 inode 加载到 inode 队列中，并使文件结构中的 fd_inode 指向它，将来介绍 inode 操作相关的函数时会再次提到。

以上描述的是 Linux 如何通过文件描述符“曲折地”找到文件数据块，聪明的您一定在暗想：“这属于有了文件描述符之后的事了，文件描述符是如何创建的呢？”其实 open 操作的本质就是创建相应文件描述符的过程，剧透一下，PCB 中文件描述符数组是提前在 task_struct 中构建好的，文件表也是提前构建好的全局数据结构，inode 队列也已经构建好了，因此笼统地说，创建文件描述符的过程就是逐层在这三个数据结构中找空位，在该空位填充好数据后返回该位置的地址，比如：

（1）在全局的 inode 队列中新建一 inode（这肯定是在空位置处新建），然后返回该 inode 地址。

（2）在全局的文件表中的找一空位，在该位置填充文件结构，使其 fd_inode 指向上一步中返回的 inode 地址，然后返回本文件结构在文件表中的下标值。

（3）在 PCB 中的文件描述符数组中找一空位，使该位置的值指向上一步中返回的文件结构下标，并返回本文件描述符在文件描述符数组中的下标值。

以上过程正是咱们要实现的。

您看，为了实现文件自由访问的需求，我们先要构建这一套逻辑，然后按照这套逻辑去访问文件，典型的自圆其说的过程，其实只要协调好就可以了。

该说的都说了，有关文件描述符的介绍就到这了，下节咱们开始动手实践。

14.3.2　文件描述符的实现

话说文件系统对外是一个子系统，但对内它是一个团队，实现新功能需要团队内部之间的合作，为了

实现文件描述符，我们不仅要添加单独的文件处理模块，还需要改进 pcb。

我们马上要做的就是创建图 14-16 左侧 pcb 中的文件描述符数组，见代码 14-7。

代码 14-7 （project/c14/c/thread/thread.h）

```
  8 #define MAX_FILES_OPEN_PER_PROC 8
…略
 77 /* 进程或线程的 pcb，程序控制块 */
 78 struct task_struct {
…略
 88    uint32_t elapsed_ticks;
 89
 90    int32_t fd_table[MAX_FILES_OPEN_PER_PROC];   // 文件描述符数组
 91
 92 /* general_tag 的作用是用于线程在一般的队列中的结点 */
 93    struct list_elem general_tag;
…略
103 };
```

fd_table 是任务的文件描述符数组，其类型是 int32_t，即每个成员都是 int32_t 整数，其长度是宏 MAX_FILES_OPEN_PER_PROC，此宏的值是 8，也就是每个任务可以打开的文件数是 8，是有点少，只要意思到了就行。

这个数组还需要被初始化，具体代码在 init_thread 函数中实现，见代码 14-8。

代码 14-8 （project/c14/c/thread/thread.c）

```
…略
 68 /* 初始化线程基本信息 */
 69 void init_thread(struct task_struct* pthread, char* name, int prio) {
…略
 88    /* 预留标准输入输出*/
 89    pthread->fd_table[0] = 0;
 90    pthread->fd_table[1] = 1;
 91    pthread->fd_table[2] = 2;
 92    /* 其余的全置为-1 */
 93    uint8_t fd_idx = 3;
 94    while (fd_idx < MAX_FILES_OPEN_PER_PROC) {
 95       pthread->fd_table[fd_idx] = -1;
 96       fd_idx++;
 97    }
 98
99     pthread->stack_magic = 0x19870916;    // 自定义的魔数
100 }
…略
```

有三个标准的文件描述符，0 是标准输入，1 是标准输出，2 是标准错误，因此在代码第 89～91 行将 fd_table[0～2]分别置为 0、1、2，也就是预留出了这 3 个文件描述符。

接着在第 93～97 行将 fd_table 中其余的文件描述符初始化为−1，在这里−1 表示该文件描述符可分配，为空位。将来会通过此值来找可分配的文件描述符。

有关线程的部分就修改这些，后面我们该介绍其他文件模块了，大伙儿下节见。

14.4 文件操作相关的基础函数

在本节我们要想实现文件及目录的创建、打开、读、写操作，必须建设好基础设施，现在咱们要一步一个脚印，慢慢走向目的地。

为了帮助大伙儿理清楚函数间的依赖关系，本文按照它们的调用关系来介绍相关的文件，同时为了减少学习的复杂性，根据实际情况有可能只会列出文件中的部分代码，并不是一股脑地把不相关的内容也搬出来，待需要的时候依然会在相同的文件中添加新功能，因此有可能同一件文件会在不同功能的讲解中反复更新，请您知晓。

14.4.1　inode 操作有关的函数

既然文件、目录本质上都是 inode，因此任何有关文件及目录的操作都离不开 inode 的处理，咱们有必要先从 inode 开始介绍，与 inode 实现相关的代码在 fs/inode.c 中，这是新创建的文件，请见代码 14-9-1。

代码 14-9-1　（project/c14/c/fs/inode.c）

```
…略
13 /* 用来存储 inode 位置 */
14 struct inode_position {
15     bool  two_sec;    // inode 是否跨扇区
16     uint32_t sec_lba;    // inode 所在的扇区号
17     uint32_t off_size;    // inode 在扇区内的字节偏移量
18 };
19
20 /* 获取 inode 所在的扇区和扇区内的偏移量 */
21 static void inode_locate(struct partition* part, uint32_t inode_no, \
   struct inode_position* inode_pos) {
22     /* inode_table 在硬盘上是连续的 */
23     ASSERT(inode_no < 4096);
       24uint32_t inode_table_lba = part->sb->inode_table_lba;
25
26     uint32_t inode_size = sizeof(struct inode);
27     uint32_t off_size = inode_no * inode_size;
       // 第 inode_no 号 I 结点相对于 inode_table_lba 的字节偏移量
28     uint32_t off_sec  = off_size / 512;
       // 第 inode_no 号 I 结点相对于 inode_table_lba 的扇区偏移量

29     uint32_t off_size_in_sec = off_size % 512;
       // 待查找的 inode 所在扇区中的起始地址
30
31     /* 判断此 i 结点是否跨越 2 个扇区 */
32     uint32_t left_in_sec = 512 - off_size_in_sec;
33     if (left_in_sec < inode_size ) {
       // 若扇区内剩下的空间不足以容纳一个 inode，必然是 I 结点跨越了 2 个扇区
34         inode_pos->two_sec = true;
35     } else {    // 否则，所查找的 inode 未跨扇区
36         inode_pos->two_sec = false;
37     }
38     inode_pos->sec_lba = inode_table_lba + off_sec;
39     inode_pos->off_size = off_size_in_sec;
40 }
41
42     /* 将 inode 写入到分区 part */
43     void inode_sync(struct partition* part, struct inode* inode, void* io_buf) {
        // io_buf 是用于硬盘 io 的缓冲区
44     uint8_t inode_no = inode->i_no;
45     struct inode_position inode_pos;
46     inode_locate(part, inode_no, &inode_pos);
       // inode 位置信息会存入 inode_pos
47     ASSERT(inode_pos.sec_lba <= (part->start_lba + part->sec_cnt));
48
49     /* 硬盘中的 inode 中的成员 inode_tag 和 i_open_cnts 是不需要的，
50      * 它们只在内存中记录链表位置和被多少进程共享 */
51     struct inode pure_inode;
52     memcpy(&pure_inode, inode, sizeof(struct inode));
53
54     /* 以下 inode 的三个成员只存在于内存中，
      现在将 inode 同步到硬盘，清掉这三项即可 */
55     pure_inode.i_open_cnts = 0;
56     pure_inode.write_deny = false;
      // 置为 false，以保证在硬盘中读出时为可写
57     pure_inode.inode_tag.prev = pure_inode.inode_tag.next = NULL;
58
59     char* inode_buf = (char*)io_buf;
60     if (inode_pos.two_sec) {
       // 若是跨了两个扇区，就要读出两个扇区再写入两个扇区

61     /* 读写硬盘是以扇区为单位，若写入的数据小于一扇区，
       要将原硬盘上的内容先读出来再和新数据拼成一扇区后再写入*/
```

```
62          ide_read(part->my_disk, inode_pos.sec_lba, inode_buf, 2);
       // inode 在 format 中写入硬盘时是连续写入的，所以读入 2 块扇区
63
64       /* 开始将待写入的 inode 拼入到这 2 个扇区中的相应位置 */
65          memcpy((inode_buf + inode_pos.off_size), \
            &pure_inode, sizeof(struct inode));
66
67       /* 将拼接好的数据再写入磁盘 */
68          ide_write(part->my_disk, inode_pos.sec_lba, inode_buf, 2);
69       } else {                       // 若只是一个扇区
70          ide_read(part->my_disk, inode_pos.sec_lba, inode_buf, 1);
71          memcpy((inode_buf + inode_pos.off_size), \
             &pure_inode, sizeof(struct inode));
72          ide_write(part->my_disk, inode_pos.sec_lba, inode_buf, 1);
73       }
74 }
75
```

程序开头定义的 struct inode_position 用于记录 inode 所在的扇区地址及在扇区内的偏移量，也就是用于定位 inode 在磁盘上的位置，其中 two_sec 用于标识 inode 是否跨扇区，即占用 2 个扇区的情况，这种情况通常出现在 inode 在扇区末创建，且扇区末尾的空间又不足矣容纳完整 inode 的时候。sec_lba 是 inode 的扇区地址，off_size 是 inode 在扇区内的偏移字节，我们后面的函数会用到此结构。

函数 inode_locate 接受 3 个参数，分区 part、inode 编号 inode_no 及 inode_pos，inode_pos 类型是上面提到的 struct inode_position，用于记录 inode 在硬盘上的位置，函数功能是定位 inode 所在的扇区和扇区内的偏移量，将其写入 inode_pos 中。

我们已经在函数 partition_format 中把 inode_table 连续地写入到磁盘上，在这之前，我们已经将分区通过函数 mount_partition 挂载了，因此可以在第 24 行从分区超级块的 inode_table_lba 中获取 inode_table 的扇区地址，存入到同名变量 inode_table_lba 中。第 26～29 行，获取编号为 inode_no 对应的 inode 的位置，off_sec 是该 inode 偏移扇区地址，off_size_in_sec 是该 inode 在扇区中的偏移字节，注意啦，off_sec 是相对于 inode_table_lba 的扇区偏移量。

inode_table 是连续写入多个扇区中的，因此在扇区的结尾处，有可能存在 inode 跨扇区的情况，即 Inode 的一部分在当前扇区末尾，另一部分在下一扇区的开头处。在第 32 行，判断此 inode 是否跨越扇区，逻辑很简单，只要判断出该 inode 所在扇区的起始地址到扇区结束之间的“剩余空间”是否大于 inode 尺寸就行了，有点拗口，这个剩余空间等于扇区大小 512 减去 inode 在扇区内的偏移字节 off_size_in_sec 的差，也就是变量 left_in_sec 的值。第 33 行判断若 left_in_sec 小于 inode_size，则将 inode_pos->two_sec 置为 true，否则置为 false。

off_sec 是相对于 inode_table 的扇区偏移量，因此 inode 的绝对扇区地址 inode_pos->sec_lba 等于 inode_table_lba 加上 off_sec，而 inode 扇区内的字节偏移量 inode_pos->off_size 仍然等于 off_size_in_sec。inode_locate 到这就介绍完了。

下面是函数 inode_sync，它接受 3 个参数，分区 part、待同步的 inode 指针、操作缓冲区 io_buf，函数功能是将 inode 写入到磁盘分区 part。

解释一下，io_buf 是主调函数提供的缓冲区，原因是必须要保证把内存数据成功同步到硬盘，同步数据需要缓冲区，一般情况下把内存中的数据同步到硬盘都是最后的操作，其前肯定已经做了大量工作，您想，若到这最后一步（也就是执行到此函数）时才申请内存失败，前面的所有操作都白费了不说，还要回滚到之前的旧状态，代价太大，因此 io_buf 必须由主调函数提前申请好。

inode_sync 的内容看上去有些多，但实际上很简单，要想往硬盘上写入 inode，必须得知道 inode 的扇区地址，也就是说得知道往硬盘哪里写，因此在第 46 行先通过函数 inode_locate 定位该 inode 的位置，位置信息在 inode_pos 中保存。

inode 中的三个成员 i_open_cnts、write_deny 和 inode_tag，它们用于统计 inode 操作状态，只在内存中有意义，为避免下次将该 inode 加载到内存时出现混乱的情况，最好是把它们写入到硬盘之前就将这三个成员的值清掉。原 inode 就不动了，这里新建一局部变量 pure_inode，第 52 行将原 inode 的值拷贝到

pure_inode 中，然后在第 55～57 行清空 pure_node 中的那 3 个变量，后续操作的都是 pure_inode。

第 59 行将 io_buf 转换为 inode_buf，此缓冲区用于拼接同步的 inode 数据。

第 60 行判断 inode 是否跨扇区，如果 inode_pos.two_sec 为 true，说明该 inode 横跨两个扇区，因此在第 62 行连续读入 2 个扇区到 inode_buf，第 65 行通过 memcpy 函数将 pure_inode 拷贝到 inode_buf 中的相应位置，第 66 行将 inode_buf 中两个扇区大小的数据写入硬盘。

第 69～72 行是处理未跨扇区的情况，原理更简单，主要区别是读写 1 个扇区的大小。

下面继续看 inode.c 的后半部分，见代码 14-9-2。

代码 14-9-2　（project/c14/c/fs/inode.c）

```
 76 /* 根据 i 结点号返回相应的 i 结点 */
 77 struct inode* inode_open(struct partition* part, uint32_t inode_no) {
 78  /* 先在已打开的 inode 链表中找 inode，此链表是为提速创建的缓冲区 */
 79     struct list_elem* elem = part->open_inodes.head.next;
 80     struct inode* inode_found;
 81     while (elem != &part->open_inodes.tail) {
 82        inode_found = elem2entry(struct inode, inode_tag, elem);
 83        if (inode_found->i_no == inode_no) {
 84           inode_found->i_open_cnts++;
 85           return inode_found;
 86        }
 87        elem = elem->next;
 88     }
 89
 90     /*由于 open_inodes 链表中找不到，
        下面从硬盘上读入此 inode 并加入到此链表 */
 91     struct inode_position inode_pos;
 92
 93     /* inode 位置信息会存入 inode_pos，
        包括 inode 所在扇区地址和扇区内的字节偏移量 */
 94     inode_locate(part, inode_no, &inode_pos);
 95
 96     /* 为使通过 sys_malloc 创建的新 inode 被所有任务共享，
 97      * 需要将 inode 置于内核空间，故需要临时
 98      * 将 cur_pbc->pgdir 置为 NULL */
 99     struct task_struct* cur = running_thread();
100     uint32_t* cur_pagedir_bak = cur->pgdir;
101     cur->pgdir = NULL;
102     /* 以上三行代码完成后下面分配的内存将位于内核区 */
103     inode_found = (struct inode*)sys_malloc(sizeof(struct inode));
104     /* 恢复 pgdir */
105     cur->pgdir = cur_pagedir_bak;
106
107     char* inode_buf;
108     if (inode_pos.two_sec) {     // 考虑跨扇区的情况
109        inode_buf = (char*)sys_malloc(1024);
110
111     /* i 结点表是被 partition_format 函数连续写入扇区的，
112      * 所以下面可以连续读出来 */
113       ide_read(part->my_disk, inode_pos.sec_lba, inode_buf, 2);
114     } else {  // 否则所查找的 inode 未跨扇区，一个扇区大小的缓冲区足够
115       inode_buf = (char*)sys_malloc(512);
116       ide_read(part->my_disk, inode_pos.sec_lba, inode_buf, 1);
117     }
118     memcpy(inode_found, inode_buf + \
        inode_pos.off_size, sizeof(struct inode));
119
120   /* 因为一会很可能要用到此 inode，故将其插入到队首便于提前检索到 */
121     list_push(&part->open_inodes, &inode_found->inode_tag);
122     inode_found->i_open_cnts = 1;
123
124     sys_free(inode_buf);
125     return inode_found;
126 }
127
128 /* 关闭 inode 或减少 inode 的打开数 */
129 void inode_close(struct inode* inode) {
130     /* 若没有进程再打开此文件，将此 inode 去掉并释放空间 */
```

```
131    enum intr_status old_status = intr_disable();
132    if (--inode->i_open_cnts == 0) {
133        list_remove(&inode->inode_tag);
                         // 将 I 结点从 part->open_inodes 中去掉
134    /* inode_open 时为实现 inode 被所有进程共享，
135     * 已经在 sys_malloc 为 inode 分配了内核空间，
136     * 释放 inode 时也要确保释放的是内核内存池 */
137      struct task_struct* cur = running_thread();
138      uint32_t* cur_pagedir_bak = cur->pgdir;
139      cur->pgdir = NULL;
140      sys_free(inode);
141      cur->pgdir = cur_pagedir_bak;
142    }
143    intr_set_status(old_status);
144 }
145
146 /* 初始化 new_inode */
147 void inode_init(uint32_t inode_no, struct inode* new_inode) {
148    new_inode->i_no = inode_no;
149    new_inode->i_size = 0;
150    new_inode->i_open_cnts = 0;
151    new_inode->write_deny = false;
152
153    /* 初始化块索引数组 i_sector */
154    uint8_t sec_idx = 0;
155    while (sec_idx < 13) {
156    /* i_sectors[12]为一级间接块地址 */
157        new_inode->i_sectors[sec_idx] = 0;
158        sec_idx++;
159    }
160 }
```

函数 inode_open 接受两个参数，分区 part 及 inode 编号 inode_no，函数功能是根据 inode_no 返回相应的 i 结点指针。

磁盘是一种低速设备，因此文件系统的设计原则是尽量减少硬盘操作。inode 是存储在磁盘上的，为减少频繁访问磁盘，我们早已在内存中为各分区创建了 inode 队列，即 part->open_inodes，很久以前就创建了，只是现在才用上。从名字上看，这个队列应该称为已打开的 inode 队列，它是 inode 的缓存。以后每打开 1 个 inode 时，先在此缓存中查找该 inode，找到后则直接返回 inode 指针，若未找到，再从磁盘上加载该 inode 到此缓存中，然后再返回其指针。查找过程就是遍历 part->open_inodes，对应的代码是第 79～88 行，如果找到后就在第 85 行执行“return inode_found”返回找到的 inode 指针。

如果 inode 队列中没有该 inode，下面开始从硬盘上读取。先在第 91 行创建 inode_pos，接着在第 94 行调用 inode_locate 去定位该 inode，位置存储到 inode_pos 中。

inode 队列中的所有 inode 应该被所有任务共享，包括内核线程和进程，这样的缓存才有意义。各进程都有独立的页表，因此为保证所有任务都共享 inode 队列，需要将整个 inode 队列创建在内核空间中。还有，为了使 inode 长久存在，inode 队列中的所有结点必须在内核的堆空间中创建，而不是简单使用栈中的局部变量。故下面我们从硬盘上获取到的 inode，其所占的内存是我们用 sys_malloc 从堆中分配的。问题终于来了，由于用户进程有自己的页表，即 pcb->pgdir 的值不为 NULL，按照 sys_malloc 的规则，这会从该用户进程自己的堆中分配内存，显然，这样做的话该 inode 只会被该进程自己访问到，失去了共享的意义。因此，为了使 inode 置于内核空间被所有任务共享，需要临时将当前任务 pcb->pgdir 置为 NULL，待为 inode 完成内存分配后再将任务的 pgdir 恢复。为简单省事，这里并未判断当前任务是内核线程，还是用户进程，采取了统一处理，因为此处判断的代价和直接赋值的代价是一样的。先在第 100 行将当前任务的页表地址备份到变量 cur_pagedir_bak 中，在第 101 行将 pgdir 置为 NULL 后，接着在第 103 行调用 sys_malloc 分配 1 个 inode 大小的内存，指针存入变量 inode_found，它是我们为磁盘上的 inode 所分配的内存变量。然后在第 105 行将 pgdir 恢复为 cur_pagedir_bak。注意，整个操作过程不是在更换页表，页表在寄存器 CR3 中呢，它可一直没有变动。这里仅仅是修改了 pcb->pgdir 而已。

inode_found 的内存分配好之后，下面开始读取硬盘了，第 108 行还是先判断该 inode 是否跨扇区，如

果是的话，就涉及到 2 个扇区的读写，因此第 109 行申请了 2 个扇区大小的缓冲区赋值给 inode_buf，之后在第 113 行读取 2 个扇区的数据到 inode_buf 中。否则 inode 未跨扇区的话，就在第 115 行只申请 1 个扇区大小的内存做缓冲区，然后将 1 扇区的数据读入到该缓冲区中。

硬盘的读写单位是扇区，我们想要的 inode 还混在这些扇区中呢，具体地址是在 inode_buf+inode_pos.off_size 处，因此在第 118 行调用 memcpy 函数将扇区中的 inode 复制到 inode_found 中。

根据程序局部性原理，通常情况下此 inode 会被再次使用到，因此在第 121 行通过 list_push 将它插入到 inode 队列的最前面，以便下次更快被找到。第 122 行将它的 i_open_cnts 置为 1，表示目前此 inode 仅被打开 1 次。然后释放缓冲区 inode_buf，返回 inode_found 指针，至此 inode_open 函数结束。

接下来是函数 inode_close，它接受 1 个参数，inode 指针 inode，功能是关闭 inode。关闭 inode 的思路是将 inode 的 i_open_cnts 减 1，若其值为 0，则说明此 inode 未被打开，此时可以将其从 inode 队列中去掉并回收空间了。相关代码是第 132～142 行。要提一点的是 inode 要先被“打开”才谈得上“关闭”，前面在用 inode_open 打开 inode 时，为了确保 inode 被所有进程共享，特意把页表置为 NULL 使内存分配函数 sys_malloc 在内核空间中为 inode 分配内存，现在释放 inode 的内存时，当前任务有可能是进程，因此会导致将内核地址在用户内存池中释放，这肯定是错的，所以第 137～139 行咱们又将页表置为 NULL，使 sys_free 正确释放内核中的 inode。sys_free 之后，再把页表换回来。

最后一个函数是 inode_init，它接受 2 个参数，inode 编号 inode_no 及待初始化的 inode 指针 new_inode，功能是初始化 new_inode。

第 148～151 行是初始化 inode 中的 i_no 为参数 inode_no，i_size 和 i_open_cnt 为 0，write_deny 为 false。

接下来是初始化 i_sectors 数组，该数组大小是 13 个元素，前 12 个是直接块地址，第 13 个一级间接块索引表地址，在此统统置为 0。inode 是由 sys_malloc 分配的，其实经 sys_malloc 返回的内存内容已被置为 0，这里为了可读性好一些还是显示初始化。

也许有同学在想，为什么不提前为 inode 分配扇区？是这样的，创建 Inode 时不为其分配扇区，只有在真正写文件时才分配扇区，理由是首先不知道文件大小，因此不知道分配多少个扇区合适，提前分配的扇区都是浪费空间。其次，文件创建后未必马上会写数据，真正需要往文件中写数据时根据实际数据量大小再分配扇区，效率更高一些。

好啦，有关 inode 操作的代码暂时只需要这么多，本节也到此结束了，以后需要时还会在 inode.c 中更新。

14.4.2　文件相关的函数

文件操作相关的函数我们定义在 fs/file.c 和 fs/file.h 中，它俩是新创建的文件，咱们还是在实践中了解吧，先介绍下头文件 file.h，见代码 14-10。

代码 14-10　（project/c14/c/fs/file.h）

```
…略
 8 /* 文件结构 */
 9 struct file {
10     uint32_t fd_pos;
   // 记录当前文件操作的偏移地址，以 0 为起始，最大为文件大小-1
11     uint32_t fd_flag;
12     struct inode* fd_inode;
13 };
14
15 /* 标准输入输出描述符 */
16 enum std_fd {
17     stdin_no,     // 0 标准输入
18     stdout_no     // 1 标准输出
19     stderr_no     // 2 标准错误
20 };
21
22 /* 位图类型 */
23 enum bitmap_type {
24     INODE_BITMAP,       // inode 位图
25     BLOCK_BITMAP        // 块位图
```

```
26 };
27
28 #define MAX_FILE_OPEN 32    // 系统可打开的最大文件数
…略
```

struct file 就是所说的文件结构，其中的 fd_pos 用于记录当前文件操作的偏移地址，该值最小是 0，最大为文件大小减 1。fd_flag 是文件操作标识，如 O_RDONLY，fd_inode 是 inode 指针，用来指向 inode 队列（part-> open_inodes）中的 inode。

enum std_fd 是标准文件描述符，在 Linux 中 0 表示标准输入，1 表示标准输出，2 表示标准错误，咱们未打算实现标准错误，因此只定义了前两个，stdin_no 表示标准输入，其值为 0，stdout_no 表示标准输出，其值为 1。

enum bitmap_type 是位图类型，包括 INODE_BITMAP 和 BLOCK_BITMAP，我们以后在往硬盘上同步位图的时候会用到此结构。

宏 MAX_FILE_OPEN 的值是 32，这是系统可打开的最大文件数。

好啦，下面介绍 file.c，见代码 14-11。

代码 14-11 （project/c14/c/fs/file.c）

```
…略
15 /* 文件表 */
16 struct file file_table[MAX_FILE_OPEN];
17
18 /*从文件表 file_table 中获取一个空闲位，成功返回下标，失败返回-1 */
19 int32_t get_free_slot_in_global(void) {
20    uint32_t fd_idx = 3;
21    while (fd_idx < MAX_FILE_OPEN) {
22       if (file_table[fd_idx].fd_inode == NULL) {
23           break;
24       }
25       fd_idx++;
26    }
27    if (fd_idx == MAX_FILE_OPEN) {
28       printk("exceed max open files\n");
29       return -1;
30    }
31    return fd_idx;
32 }
33
34 /* 将全局描述符下标安装到进程或线程自己的文件描述符数组 fd_table 中，
35  * 成功返回下标，失败返回-1 */
36 int32_t pcb_fd_install(int32_t globa_fd_idx) {
37    struct task_struct* cur = running_thread();
38    uint8_t local_fd_idx = 3;  // 跨过 stdin,stdout,stderr
39    while (local_fd_idx < MAX_FILES_OPEN_PER_PROC) {
40       if (cur->fd_table[local_fd_idx] == -1) {  // -1 表示 free_slot，可用
41           cur->fd_table[local_fd_idx] = globa_fd_idx;
42           break;
43       }
44       local_fd_idx++;
45    }
46    if (local_fd_idx == MAX_FILES_OPEN_PER_PROC) {
47        printk("exceed max open files_per_proc\n");
48        return -1;
49    }
50    return local_fd_idx;
51 }
52
53 /* 分配一个 i 结点，返回 i 结点号 */
54 int32_t inode_bitmap_alloc(struct partition* part) {
55    int32_t bit_idx = bitmap_scan(&part->inode_bitmap, 1);
56    if (bit_idx == -1) {
57        return -1;
58    }
59    bitmap_set(&part->inode_bitmap, bit_idx, 1);
60    return bit_idx;
61 }
62
63 /* 分配 1 个扇区，返回其扇区地址 */
```

```
64 int32_t block_bitmap_alloc(struct partition* part) {
65    int32_t bit_idx = bitmap_scan(&part->block_bitmap, 1);
66    if (bit_idx == -1) {
67        return -1;
68    }
69    bitmap_set(&part->block_bitmap, bit_idx, 1);
70    /* 和 inode_bitmap_malloc 不同，此处返回的不是位图索引，
      而是具体可用的扇区地址 */
71    return (part->sb->data_start_lba + bit_idx);
72 }
73
74 /* 将内存中 bitmap 第 bit_idx 位所在的 512 字节同步到硬盘 */
75 void bitmap_sync(struct partition* part, uint32_t bit_idx, uint8_t btmp) {
76    uint32_t off_sec = bit_idx / 4096;
                        // 本 i 结点索引相对于位图的扇区偏移量
77    uint32_t off_size = off_sec * BLOCK_SIZE;
                        // 本 i 结点索引相对于位图的字节偏移量
78    uint32_t sec_lba;
79    uint8_t* bitmap_off;
80
81 /* 需要被同步到硬盘的位图只有 inode_bitmap 和 block_bitmap */
82    switch (btmp) {
83       case INODE_BITMAP:
84          sec_lba = part->sb->inode_bitmap_lba + off_sec;
85          bitmap_off = part->inode_bitmap.bits + off_size;
86          break;
87
88       case BLOCK_BITMAP:
89          sec_lba = part->sb->block_bitmap_lba + off_sec;
90          bitmap_off = part->block_bitmap.bits + off_size;
91          break;
92    }
93    ide_write(part->my_disk, sec_lba, bitmap_off, 1);
94 }
…略
```

代码中的 file_table 是文件表，也就是文件结构数组，它的长度是 MAX_FILE_OPEN，也就是最多可同时打开 MAX_FILE_OPEN 次文件，由于一个文件可以被多次打开，甚至把 file_table 占满，故此处用的单位是“次”，而不是“个”（想想这么严谨也是醉了）。

函数 get_free_slot_in_global 功能是从文件表 file_table 中获取一个空闲位，成功则返回空闲位下标，失败则返回−1。实现原理是遍历 file_table，找出 fd_inode 为 null 的数组元素，该元素表示为空，将其下标返回即可。另外，file_table 中的前 3 个成员预留给标准输入、标准输出及标准错误，以后需要用到它们。

函数 pcb_fd_install 接受 1 个参数，全局描述符下标 globa_fd_idx，也就是数组 file_table 的下标。函数功能是将 globa_fd_idx 安装到进程或线程自己的文件描述符数组 fd_table 中，成功则返回 fd_table 中空位的下标，失败则返回−1。位于 pcb 中的 fd_table，前 3 个元素是标准文件描述符，分别表示标准输入、标准输出和标准错误，不能占用，其余的都是可分配的描述符，我们已在初始化线程时将这些可分配的描述符置为−1。因此只要 fd_table 数组元素为−1 就表示空位、可分配的文件描述符，具体实现是第 38～50 行代码，很简单，不多说了。

函数 inode_bitmap_alloc，它接受 1 个参数，分区 part，功能是分配一个 i 结点，返回 i 结点号。位图操作大伙儿都已经清楚了，不多说了。

函数 block_bitmap_alloc 接受 1 个参数，分区 part，功能是分配 1 个扇区，返回其扇区地址。其实现也是位图操作，很简单不说了。

函数 bitmap_sync 接受 3 个参数，分区 part、位索引 bit_idx、位图类型 btmp_type，功能是将内存中 bitmap 第 bit_idx 位所在的 512 字节同步到硬盘。因为硬盘是以扇区为读写单位，所以内存中的数据也要一次操作 1 扇区，函数开头要确定待同步的位属于哪一个 512 字节中，第 76 行用 bit_idx/4096 计算第 bit_idx 位相对于位图的以扇区为单位的偏移量，结果存入 off_sec，也就是说第 bit_idx 位属于位图中第 off_sec 个扇区大小的内存中，不过 off_sec 用于计算写入硬盘的扇区地址，下一行将 off_sec 乘以 BLOCK_SIZE 获得该位相对于位图的字节偏移量，结果存入 off_size，off_size 才用于待写入硬盘的内存数据，它是第 bit_idx

位所在位图中以 512 字节为单位的起始地址。文件系统中目前有两种位图，inode 位图及块位图，下面的 switch 结构通过位图类型 btmp_type 来判断同步哪种位图，分别计算出位图的扇区地址 sec_lba 和字节偏移量 bitmap_off，bitmap_off 是待同步数据的起始地址，最后通过 ide_write 将位图写入硬盘。

将来咱们还会在 file.c 中新建一些代码，用到的时候再讨论吧，本节先到这，下节再见。

14.4.3 目录相关的函数

目录对文件系统来说是非常重要的概念，目录大家都知道，相当于普通文件的容器，为支持目录操作，咱们先定义好一些基础方法。有关目录操作的函数我们定义在 fs/dir.c 中，好啦，上菜喽，见代码 14-12-1。

代码 14-12-1 （project/c14/c/fs/dir.c）

```
…略
14 struct dir root_dir;             // 根目录
15
16 /* 打开根目录 */
17 void open_root_dir(struct partition* part) {
18     root_dir.inode = inode_open(part, part->sb->root_inode_no);
19     root_dir.dir_pos = 0;
20 }
21
22     /* 在分区 part 上打开 i 结点为 inode_no 的目录并返回目录指针 */
23     struct dir* dir_open(struct partition* part, uint32_t inode_no) {
24     struct dir* pdir = (struct dir*)sys_malloc(sizeof(struct dir));
25     pdir->inode = inode_open(part, inode_no);
26     pdir->dir_pos = 0;
27     return pdir;
28 }
29
30     /* 在 part 分区内的 pdir 目录内寻找名为 name 的文件或目录，
31      * 找到后返回 true 并将其目录项存入 dir_e，否则返回 false */
32     bool search_dir_entry(struct partition* part, struct dir* pdir, \
33                           const char* name, struct dir_entry* dir_e) {
34     uint32_t block_cnt = 140;  // 12 个直接块+128 个一级间接块=140 块
35
36     /* 12 个直接块大小+128 个间接块,共 560 字节 */
37     uint32_t* all_blocks = (uint32_t*)sys_malloc(48 + 512);
38     if (all_blocks == NULL) {
39          printk("search_dir_entry: sys_malloc for all_blocks failed");
40          return false;
41     }
42
43     uint32_t block_idx = 0;
44     while (block_idx < 12) {
45        all_blocks[block_idx] = pdir->inode->i_sectors[block_idx];
46        block_idx++;
47     }
48     block_idx = 0;
49
50     if (pdir->inode->i_sectors[12] != 0) {    // 若含有一级间接块表
51        ide_read(part->my_disk, \
      pdir->inode->i_sectors[12], all_blocks + 12, 1);
52     }
53     /* 至此，all_blocks 存储的是该文件或目录的所有扇区地址 */
54
55     /* 写目录项的时候已保证目录项不跨扇区，
56      * 这样读目录项时容易处理，只申请容纳 1 个扇区的内存 */
57     uint8_t* buf = (uint8_t*)sys_malloc(SECTOR_SIZE);
58     struct dir_entry* p_de = (struct dir_entry*)buf;
       // p_de 为指向目录项的指针，值为 buf 起始地址
59     uint32_t dir_entry_size = part->sb->dir_entry_size;
60     uint32_t dir_entry_cnt = SECTOR_SIZE / dir_entry_size;
       // 1 扇区内可容纳的目录项个数
61
62    /* 开始在所有块中查找目录项 */
63     while (block_idx < block_cnt) {
64     /* 块地址为 0 时表示该块中无数据，继续在其他块中找 */
65        if (all_blocks[block_idx] == 0) {
```

```
66            block_idx++;
67            continue;
68        }
69        ide_read(part->my_disk, all_blocks[block_idx], buf, 1);
70
71        uint32_t dir_entry_idx = 0;
72        /* 遍历扇区中所有目录项 */
73        while (dir_entry_idx < dir_entry_cnt) {
74            /* 若找到了，就直接复制整个目录项 */
75            if (!strcmp(p_de->filename, name)) {
76                memcpy(dir_e, p_de, dir_entry_size);
77                sys_free(buf);
78                sys_free(all_blocks);
79                return true;
80            }
81            dir_entry_idx++;
82            p_de++;
83        }
84        block_idx++;
85        p_de = (struct dir_entry*)buf;
       // 此时 p_de 已经指向扇区内最后一个完整目录项，
                    // 需要恢复 p_de 指向为 buf
86        memset(buf, 0, SECTOR_SIZE);     // 将 buf 清 0，下次再用
87    }
88    sys_free(buf);
89    sys_free(all_blocks);
90    return false;
91 }
```

程序开头定义了 root_dir，它是分区的根目录。

函数 open_root_dir 接受一个参数，分区 part，功能是打开分区 part 的根目录。实现非常简单，先调用 inode_open 打开根目录的 inode，然后将其赋值给根目录的 inode 指针。随后将根目录的 dir_pos 置为 0。好啦，完成。

接下来是函数 dir_open，它接受两个参数，分区 part 和 inode 编号 inode_no，功能是在分区 part 上打开 i 结点为 inode_no 的目录并返回目录指针。此函数与 open_root_dir 类似，区别是根目录 root_dir 是提前定义好的全局变量，免去了为其申请内存的过程，这里要为其他目录单独申请内存，也就是第 24 行代码的功能。其他都一样，不说了。

下面的函数 search_dir_entry 可是个大活儿，内容有点多，它接受 4 个参数，分区 part、目录指针 pdir、文件名 name、目录项指针 dir_e，函数功能是在 part 分区内的 pdir 目录内寻找名为 name 的文件或目录，找到后返回 true 并将其目录项存入 dir_e，否则返回 false。

函数开头定义了变量 block_cnt，表示 inode 总的块数，其值为 140，即 12 个直接块+128 个一级间接块。接下来第 37 行为这 140 个扇区地址申请内存，返回地址赋值给 all_blocks，这样做是为了方便检索此 inode 的全部扇区地址，我们以后对此目录 inode 所在的扇区地址都统一从 all_blocks 中获取，因此在第 43～47 行，先将目录 inode 的 i_sectors 中的前 12 个扇区地址录入到 all_blocks 中，随后在第 50 行判断是否有一级间接块索引表，只要 i_sectors[12]不为 0 就表示有一级间接块索引表，如果有，就从硬盘的扇区地址 i_sectors[12]处获取 1 扇区数据，此数据是 128 个间接块地址，将其复制到 all_blocks+12 字节处。至此，all_blocks 被写满了，其存储的是目录 pdir 的所有扇区地址。

我们在处理 inode_table 时，是将它连续写在多个扇区中，也就是除最后一个扇区外，其他几个扇区都写满了 inode，从而导致了 inode 跨扇区的情况，以至于在获取 inode 的时候要做额外判断，比较麻烦。吸取经验教训，我们在往目录中写目录项的时候，写入的都是完整的目录项，避免了目录项跨扇区的情况，因此在实际搜索目录项的时候每次只从硬盘读取一扇区就好了，所以在第 57 行，我们为缓冲区 buf 申请的内存大小是 SECTOR_SIZE，即 1 扇区。第 58 行将缓冲区转换为目录项 struct dir_entry 类型，赋值给 p_de，后面将开始用 p_de 遍历 buf。第 59～60 行计算一扇区内容纳的目录项数，结果存入变量 dir_entry_cnt。

由于我们不知道目录项在目录的哪个扇区中存在，所以在第 63 行的 while 循环开始，我们在该目录所有的扇区中查找目录项。目录的所有扇区地址已经被收录到 all_blocks 中，在第 65 行判断，如果扇区地

址为 0，这说明未分配扇区，跳过，继续下一次循环。这里要说明一下，尽管我们将来往目录中写目录项时，是优先在 inode->i_sectors 扇区数组中靠前的扇区中写入，前面的扇区无法容纳完整目录项后才会占用新的扇区。但将来我们还会实现删除文件和目录的功能，删除文件的顺序可是随机的，这取决于文件所在的扇区。若目录中的文件或子目录过多，占用了多个扇区，执行删除文件时，有可能被删除的文件位于前面扇区中，后面扇区中的文件还在，比如目录 a 中先生成了 10 个文件，假如这 10 个文件位于扇区地址 i_sector[0]处，正巧这个扇区中只能容纳 10 个目录项，后来又在目录 a 中生成了第 11 个文件，该文件占用的扇区地址是 i_sector[1]，现在将前 10 个文件全部删除，删除后，扇区 i_sector[0]上已无文件，因此将 i_sector[0]回收，那时会将 i_sector[0]赋值为 0，但目录 a 的扇区地址 i_sector[1]不为 0，该扇区中还有文件，所以扇区地址为 0 时还要继续在后面的索引块中查找，不能放弃。

回到代码，若在第 65 行判断 all_blocks[block_idx]不为 0，这表示已分配扇区地址了，于是在第 69 行从该扇区地址 all_blocks[block_idx]读入 1 扇区数据到 buf，在第 73～83 行用目录项指针 p_de 遍历该扇区内的所有目录项，在第 75 行比较目录项的 p_de->filename 是否和待查找的文件名 name 相等，若相等则表示找到该文件，然后第 76 行将该目录项 p_de 复制到参数 dir_e 中，之后释放缓冲区和 all_blocks，成功返回。否则，在第 81～82 行更新为下一个目录项继续在该扇区中查找。若该扇区中未找到该文件的话，在第 84 行使 block_idx++，更新为 all_blocks 中的下一个扇区，读取新的扇区，重复以上查找过程。如果到最后都未找到该文件，在第 88～90 行释放 buf 和 all_blocks，返回 false，至此函数 search_dir_entry 结束。

下面看 dir.c 的中间部分，见代码 14-12-2。

代码 14-12-2　（project/c14/c/fs/dir.c）

```
…略
 93 /* 关闭目录 */
 94 void dir_close(struct dir* dir) {
 95 /*************      根目录不能关闭     ***************
 96   *1 根目录自打开后就不应该关闭，否则还需要再次 open_root_dir();
 97   *2 root_dir 所在的内存是低端 1MB 之内，并非在堆中，free 会出问题 */
 98     if (dir == &root_dir) {
 99    /* 不做任何处理直接返回*/
100        return;
101     }
102     inode_close(dir->inode);
103     sys_free(dir);
104 }
105
106 /*在内存中初始化目录项 p_de */
107 void create_dir_entry(char* filename, uint32_t inode_no, \
    uint8_t file_type, struct dir_entry* p_de) {
108     ASSERT(strlen(filename) <=  MAX_FILE_NAME_LEN);
109
110     /* 初始化目录项 */
111     memcpy(p_de->filename, filename, strlen(filename));
112     p_de->i_no = inode_no;
113     p_de->f_type = file_type;
114 }
…略
```

这部分很短小，就两个功能。

函数 dir_close 接受 1 个参数，目录指针 dir，功能是关闭目录 dir。关闭目录的本质是关闭目录的 inode 并释放目录占用的内存，这分别是通过第 102～103 行的代码实现的。不过有两条注释说明了我们对根目录做了特殊处理，根目录不能被真正地关闭，不做任何处理直接返回。原因是首先根目录始终应该是打开的，它是所有目录的父目录，查找文件时必须要从根目录开始找。其次是根目录 root_dir 占用的是静态内存，它位于低端 1MB 之内，并非是在堆中申请的，不能将其释放。

函数 create_dir_entry 接受 4 个参数，文件名 filename、inode 编号 inode_no、文件类型 file_type、目录项指针 p_de。功能是在内存中创建目录项 p_de。函数的实现就是在初始化目录项 p_de：将文件名拷贝到目录项 p_de->filename 中，用 inode_no 为 p_de->i_no 赋值，用 file_type 为 p_de->f_type 赋值。

下面看 dir.c 的最后部分，见代码 14-12-3。

代码 14-12-3　（project/c14/c/fs/dir.c）

```
…略
116 /* 将目录项 p_de 写入父目录 parent_dir 中，io_buf 由主调函数提供 */
117 bool sync_dir_entry(struct dir* parent_dir, \
    struct dir_entry* p_de, void* io_buf) {
118    struct inode* dir_inode = parent_dir->inode;
119    uint32_t dir_size = dir_inode->i_size;
120    uint32_t dir_entry_size = cur_part->sb->dir_entry_size;
121
122    ASSERT(dir_size % dir_entry_size == 0);
                          // dir_size 应该是 dir_entry_size 的整数倍
123
124    uint32_t dir_entrys_per_sec = (512 / dir_entry_size);
                          // 每扇区最大的目录项数目
125    int32_t block_lba = -1;
126
127    /* 将该目录的所有扇区地址
       （12 个直接块+ 128 个间接块）存入 all_blocks */
128    uint8_t block_idx = 0;
129    uint32_t all_blocks[140] = {0};   // all_blocks 保存目录所有的块
130
131    /* 将 12 个直接块存入 all_blocks */
132    while (block_idx < 12) {
133       all_blocks[block_idx] = dir_inode->i_sectors[block_idx];
134       block_idx++;
135    }
136
137    struct dir_entry* dir_e = (struct dir_entry*)io_buf;
                                           // dir_e 用来在 io_buf 中遍历目录项
138    int32_t block_bitmap_idx = -1;
139
140    /* 开始遍历所有块以寻找目录项空位，若已有扇区中没有空闲位，
141     * 在不超过文件大小的情况下申请新扇区来存储新目录项 */
142    block_idx = 0;
143    while (block_idx < 140) {
       // 文件（包括目录）最大支持 12 个直接块+128 个间接块＝140 个块

144       block_bitmap_idx = -1;
145       if (all_blocks[block_idx] == 0) {   // 在三种情况下分配块
146          block_lba = block_bitmap_alloc(cur_part);
147          if (block_lba == -1) {
148             printk("alloc block bitmap for sync_dir_entry failed\n");
149             return false;
150          }
151
152       /* 每分配一个块就同步一次 block_bitmap */
153          block_bitmap_idx = block_lba - cur_part->sb->data_start_lba;
154          ASSERT(block_bitmap_idx != -1);
155          bitmap_sync(cur_part, block_bitmap_idx, BLOCK_BITMAP);
156
157          block_bitmap_idx = -1;
158          if (block_idx < 12) {     // 若是直接块
159             dir_inode->i_sectors[block_idx] = \
                all_blocks[block_idx] = block_lba;
160          } else if (block_idx == 12) {
             // 若是尚未分配一级间接块表（block_idx 等于 12 表示第 0 个间接块地址为 0）

161             dir_inode->i_sectors[12] = block_lba;
                // 将上面分配的块作为一级间接块表地址
162             block_lba = -1;
163             block_lba = block_bitmap_alloc(cur_part);
                // 再分配一个块作为第 0 个间接块
164             if (block_lba == -1) {
165                  block_bitmap_idx = dir_inode->i_sectors[12] - \
                     cur_part->sb->data_start_lba;
166                  bitmap_set(&cur_part->block_bitmap, \
                      block_bitmap_idx, 0);
167                  dir_inode->i_sectors[12] = 0;
168                  printk("alloc block bitmap for sync_dir_entry failed\n");
169                  return false;
170             }
```

```
171
172             /* 每分配一个块就同步一次 block_bitmap */
173                 block_bitmap_idx = block_lba - cur_part->sb->data_start_lba;
174                 ASSERT(block_bitmap_idx != -1);
175                 bitmap_sync(cur_part, block_bitmap_idx, BLOCK_BITMAP);
176
177                 all_blocks[12] = block_lba;
178                 /* 把新分配的第 0 个间接块地址写入一级间接块表 */
179                 ide_write(cur_part->my_disk, \
                     dir_inode->i_sectors[12], all_blocks + 12, 1);
180             } else {        // 若是间接块未分配
181                 all_blocks[block_idx] = block_lba;
182       /* 把新分配的第(block_idx-12)个间接块地址写入一级间接块表 */
183                 ide_write(cur_part->my_disk, \
                     dir_inode->i_sectors[12], all_blocks + 12, 1);
184             }
185
186             /* 再将新目录项 p_de 写入新分配的间接块 */
187             memset(io_buf, 0, 512);
188             memcpy(io_buf, p_de, dir_entry_size);
189             ide_write(cur_part->my_disk, all_blocks[block_idx], io_buf, 1);
190             dir_inode->i_size += dir_entry_size;
191             return true;
192         }
193
194      /* 若第 block_idx 块已存在，将其读进内存，\
         然后在该块中查找空目录项 */
195         ide_read(cur_part->my_disk, all_blocks[block_idx], io_buf, 1);
196         /* 在扇区内查找空目录项 */
197         uint8_t dir_entry_idx = 0;
198         while (dir_entry_idx < dir_entrys_per_sec) {
199             if ((dir_e + dir_entry_idx)->f_type == FT_UNKNOWN) {
              // FT_UNKNOWN 为 0，无论是初始化，或是删除文件后，
                 // 都会将 f_type 置为 FT_UNKNOWN
200                 memcpy(dir_e + dir_entry_idx, p_de, dir_entry_size);
201                 ide_write(cur_part->my_disk, all_blocks[block_idx], io_buf, 1);
202
203                 dir_inode->i_size += dir_entry_size;
204                 return true;
205             }
206             dir_entry_idx++;
207         }
208         block_idx++;
209      }
210      printk("directory is full!\n");
211      return false;
212 }
```

也许此时您还沉浸在代码 14-12-2 内容较少的欣慰中，现在该醒醒了，这最后一部分代码中只有函数 sync_dir_entry，它的内容很多，因此把它放在最后介绍。

sync_dir_entry 接受 3 个参数，父目录 parent_dir、目录项 p_de、缓冲区 io_buf，功能是将目录项 p_de 写入父目录 parent_dir 中，其中 io_buf 由主调函数提供。

当 inode 是目录时，其 i_size 是目录中目录项的大小之和，父目录的大小是 dir_inode->i_size，第 119 行将其赋值给变量 dir_size。目录项大小记录在超级块中，第 120 行获取了超级块的大小，存入变量 dir_entry_size。第 124 行计算 1 扇区可容纳的完整的目录项数，结果写入变量 dir_entrys_per_sec，此处就是在函数 search_dir_entry 中所说的：写入目录项时已保证目录项不会跨扇区。

在第 128～135 行，将目录的 12 个直接块地址收集到数组 all_blocks，下面将优先检查这 12 个扇区。第 137 行使目录项指针 dir_e 指向缓冲区 io_buf，这里为目录项指针变量起名为 dir_e 是避免与参数 p_de 冲突。

由于删除文件时会造成目录中存在空洞，也就是文件系统内的碎片，所以在写入文件时，要逐个目录项查找空位，避免一味在目录的末尾添加目录项，而前面所有文件已被删除时，却还占用多个扇区的情况，所以第 143 行开始从头在这 12 个扇区中找空闲目录项位置。虽然我们只在 all_blocks 中收集了 12 个直接块的地址，但依然要遍历 140 个扇区，因为要是在这 12 个扇区中找不到目录项空位时，在文件大小未超

过 140 个扇区的情况下，我们还会为其分配新扇区以容纳目录项。代码第 145 行先判断扇区是否分配，若未分配，在第 146 行通过函数 block_bitmap_alloc 为其分配一扇区，扇区地址写入变量 block_lba。由于 block_bitmap_alloc 仅是操作内存中的块位图，为保持数据同步，现在要将块位图同步到硬盘，于是在第 153 行计算 block_lba 相对于 data_start_lba 的偏移，第 155 行调用 bitmap_sync 将块位图同步到硬盘。

第 158 行判断当前为空的块是直接块，还是间接块，若块索引小于 12，则属于直接块，故在第 159 行将分配的扇区地址写入 i_sectors[block_idx]和 all_blocks[block_idx]。若正好是第 12 个块，即一级间接块索引表地址为空，下面就该创建间接块了，在第 161 行将刚才分配的扇区地址 block_lba 作为一级间接块索引表的地址写入 i_sectors[12]，在第 163 重新再分配一扇区，此时 block_lba 更新为新分配的扇区地址，该扇区地址将作为第 0 个间接块。如果 block_lba 为−1，也就是分配扇区失败了，在第 165～169 行执行一些回滚操作。如果分配成功的话，接着在第 173～175 行将块位图同步到硬盘。然后在第 177 行将新分配的扇区地址更新到 all_blocks[12]，这是第 0 个间接块的地址，随后在第 179 行调用 ide_write 将间接块地址写入一级间接块索引表所在的扇区。

在第 180 行，若 block_idx 已经超过了 12，这说明已经遍历到间接块了，将最初分配的扇区地址 block_lba 录入 all_blocks，然后在第 183 行将间接块地址写入一级间接块索引表所在的扇区。最后在第 187～189 行将目录项写入扇区，然后第 190 行更新目录大小，使目录的 i_size 加上 1 个目录项大小 dir_entry_size。

第 195 行开始处理块已存在，不需要分配块的情况，也就是要在该扇区中寻找空闲的目录项，先在第 195 行将该扇区读到 io_buf 中，接着在第 198 行通过 while 循环遍历 dir_entrys_per_sec 个目录项，第 199 行判断若目录项的 f_type 为 FT_UNKNOWN，这表示该目录项未分配，可以用，于是在第 200 行将目录项 p_de 写入 io_buf，接着调用 ide_write 将目录项同步到硬盘，最后使目录的 i_size 加上 1 个目录项大小 dir_entry_size。如果所有的扇区都满了，直接输出“directory is full!”并以 false 返回。

好啦，dir.c 目前就更新到这，以后需要时再添加新功能，兄弟们真的辛苦了，下节再见。

14.4.4　路径解析相关的函数

本节咱们要完成路径解析的功能。路径大家都清楚，比如 Windows 中的“C:\windows”，Linux 中的“/home”。在这两种路径中都有路径分隔符，Window 中的路径分隔符是'\'，Linux 的是'/'，咱们采取和 Linux 一样的路径规则。

什么是路径解析呢？简单来说，就是把路径按照路径分隔符拆分成多层文件名，逐层在磁盘上查找以确认文件名是否存在。比如路径“/a/b/c”会被分别拆分成“a”，“b”，“c”。Linux 中一切皆文件，因此同一目录下不能出现同名的普通文件和目录。从路径上咱们可以猜出“a”和“b”都是目录（但也有可能“a”只是个普通文件，用户误把它当成目录），“c”可能是普通文件，也可能是目录。不管猜的对不对，文件系统必须逐层到目录项中去确认。最左边的'/'表示根目录，首先在根目录下查找 a，根据目录项的文件类型，发现它是目录，继续在 a 目录中查找 b，同样发现其是目录，然后在 b 目录中查找 c，c 也许是普通文件，也可能是目录，这还是要根据目录项来判断。当然了，在逐层查找的过程中，也许某层路径就不存在了，比如在 a 目录中未找到文件 b 时，后面的子路径 c 就不需要再查了。

大伙儿不要觉得只有在查找文件时才需要路径解析，路径解析是在任何时刻都需要的功能，比如命令“cd /a/b”，也要先从根目录/开始，逐层解析 a 和 b 目录。有关路径解析的代码是在 fs.c 中，咱们从简入深分别介绍，见代码 14-13。

代码 14-13　（project/c14/c/fs/fs.c）

```
…略
192 /* 将最上层路径名称解析出来 */
193 static char* path_parse(char* pathname, char* name_store) {
194     if (pathname[0] == '/') {   // 根目录不需要单独解析
195      /* 路径中出现 1 个或多个连续的字符'/'，将这些'/'跳过，如"///a/b" */
196         while(*(++pathname) == '/');
197     }
198
199     /* 开始一般的路径解析 */
```

```
200     while (*pathname != '/' && *pathname != 0) {
201         *name_store++ = *pathname++;
202     }
203
204     if (pathname[0] == 0) {   // 若路径字符串为空，则返回 NULL
205         return NULL;
206     }
207     return pathname;
208 }
209
210 /* 返回路径深度，比如/a/b/c，深度为 3 */
211 int32_t path_depth_cnt(char* pathname) {
212     ASSERT(pathname != NULL);
213     char* p = pathname;
214     char name[MAX_FILE_NAME_LEN];
                            // 用于 path_parse 的参数做路径解析
215     uint32_t depth = 0;
216
217     /* 解析路径，从中拆分出各级名称 */
218     p = path_parse(p, name);
219     while (name[0]) {
220         depth++;
221         memset(name, 0, MAX_FILE_NAME_LEN);
222         if (p) {      // 如果 p 不等于 NULL，继续分析路径
223             p  = path_parse(p, name);
224         }
225     }
226     return depth;
227 }
…略
```

path_parse 函数接受 2 个参数，pathname 是字符串形式的路径及文件名，name_store 是主调函数提供的缓冲区，用于存储最上层路径名，此函数功能是将最上层路径名称解析出来存储到 name_store 中，调用结束后返回除顶层路径之外的子路径字符串的地址。每次调用 path_parse 只会解析一层路径名，也就是最顶层的路径名，例如要是解析路径“/a/b/c”的话，调用 path_parse 时，name_store 的值是“a”，然后返回子路径字符串“/b/c”的地址。下次再调用 path_parse 时，主调函数若传给 pathname 的参数是“/b/c”，path_parse 执行后 name_store 中的值变为“b”，然后返回子路径“/c”的地址。

函数开头先判断路径名第 0 个字符是否为'/'，也就是说类似这样的路径“/a/b/c”。首先，任何时候路径中最左边的'/'都表示根目录，根目录不需要单独解析，因为它是已知的，并且已经被提前打开了。其次，路径中的'/'仅表示路径分隔符，我们并不关心它，只关心路径分隔符之间的路径名，因此无论'/'表示根目录，还是路径分隔符，我们始终要跨过路径中最左边的'/'。

接下来在第 196 行通过 while 循环去掉路径中连续重复的多个'/'，也许您可能会想，路径中的各层目录间不是只有一个'/'吗，顶多去掉 1 个'/'就可以了，去掉多个'/'是什么情况？确实，一般情况下我们输入的路径都是以 1 个'/'作为分隔符，但在 Linux 中各层路径间的分隔符可以是多个'/'，比如"//a/b///c"这样的路径是合法的，我们也兼容此方式，因此必须要去掉路径中连续重复的多个'/'。

第 200 行的 while 循环开始路径解析，也就是解析出参数 pathname 中最顶层（最左边）的路径名存入 name_store。比如若此时的 pathname 经过 196 行的处理后已变为“a/b”的话，循环处理后，name_store 为 a。

若 pathname 已经结束，指向末尾的结束字符'\0'（其 ASCII 值为 0），就返回 NULL 表示处理结束，否则 pathname 依然包含子路径，将其返回。

函数 path_depth_cnt 接受 1 个参数，pathname 表示待分析的路径。函数功能是返回路径深度，比如路径“/a/b/c”，深度为 3。不过，这里不能简单地根据路径分隔符'/'来统计目录深度，前面说过在 Linux 中"//a/b///c"这样的路径是合法的，我们也是如此。函数实现较简单，主要是循环调用 path_parse 统计各级路径名，通过变量 depth 来累计路径层数，不多说了。

还有个文件搜索的函数未介绍，该函数有占长，咱们单独拿出来介绍吧，本节到此结束，大伙儿下节再会。

14.4.5　实现文件检索功能

在打开文件之前，咱们要确认文件是否在磁盘上存在，毕竟只有已存在的文件才谈得上“打开”。同样在创建文件之前，文件系统也要确认被创建的文件所在的目录中是否已有同名文件存在，毕竟目录下不允许存在同名的文件。您看，这实际上就是文件搜索的功能，本节继续完善 fs.c，在其中添加函数 search_file。

在头文件 fs.h 中又更新了一些结构，咱们先看看里面有什么，见代码 14-14。

代码 14-14　（project/c14/c/fs/fs.h）

```
…略
10 #define MAX_PATH_LEN 512      // 路径最大长度
11
12 /* 文件类型 */
13 enum file_types {
14     FT_UNKNOWN,     // 不支持的文件类型
15     FT_REGULAR,     // 普通文件
16     FT_DIRECTORY    // 目录
17 };
18
19 /* 打开文件的选项 */
20 enum oflags {
21     O_RDONLY,       // 只读
22     O_WRONLY,       // 只写
23     O_RDWR,         // 读写
24     O_CREAT = 4     // 创建
25 };
26
27 /* 用来记录查找文件过程中已找到的上级路径，
   也就是查找文件过程中“走过的地方” */
28 struct path_search_record {
29     char searched_path[MAX_PATH_LEN];     // 查找过程中的父路径
30     struct dir* parent_dir;     // 文件或目录所在的直接父目录
31     enum file_types file_type;
       // 找到的是普通文件，还是目录，找不到将为未知类型(FT_UNKNOWN)
32 };
…略
```

宏 MAX_PATH_LEN 表示路径名最大的长度，这里其值为 512。

枚举结构 enum oflags 是打开文件时的选项，将来实现 file_create 和 open 函数时会用到，提前就放这了，要不到时候就为这几行定义再重复说明 fs.h 真是不值当的。简单说一下 flags，查看 open 函数的帮助您就会知道，其原型是“int open(const char *pathname, int flags)”，它提供一些标识选项，也就是参数 flags，可选的值一般有 O_RDONLY、O_WRONLY、O_RDWR、O_CREAT、O_EXCL 等，它们是定义在文件“/usr/include/asm-generic/fcntl.h”中的宏，如图 14-17 所示。

```
 6 /* open/fcntl - O_SYNC is only implemented on blocks devices and on files
 7    located on an ext2 file system */
 8 #define O_ACCMODE       00000003
 9 #define O_RDONLY        00000000
10 #define O_WRONLY        00000001
11 #define O_RDWR          00000002
12 #ifndef O_CREAT
13 #define O_CREAT         00000100        /* not fcntl */
14 #endif
15 #ifndef O_EXCL
16 #define O_EXCL          00000200        /* not fcntl */
17 #endif
18 #ifndef O_NOCTTY
19 #define O_NOCTTY        00000400        /* not fcntl */
20 #endif
21 #ifndef O_TRUNC
22 #define O_TRUNC         00001000        /* not fcntl */
23 #endif
24 #ifndef O_APPEND
25 #define O_APPEND        00002000
```

▲图 14-17　flags 定义

目前咱们 enum oflags 结构中只支持 4 个标识，这里是按“位”来定义各标识的，按二进制来说，O_RDONLY 的值为 000b，O_WRONLY 的值为 001b，O_RDWR 的值为 010b，O_CREAT 的值为 100b，这样的好处是当这几个标识通过位或“|”叠加到一起作为复合参数时，可以通过位与运算“&”单独解析出各位以反推标识位。

下面定义了 struct path_search_record，它是路径搜索记录，此结构用来记录查找文件过程中已处理过的上级路径，也就是查找文件过程中"走过的地方。用此结构的目的是想获取路径中“断链”的部分，比如查找文件“/a/b/c”，若找不到的话，我们想知道是 c 不存在，还是上层目录 b 或 a 就不存在，其中成员 searched_path 就是查找过程中不存在的路径，还是拿查找“/a/b/c”来说，若仅是 c 不存在，searched_path 的值

为“/a/b/c”，若是 b 就不存在，searched_path 的值为“/a/b”。成员 parent_dir 用于记录文件或目录所在的直接父目录，成员 file_type 是找到的文件类型，若找不到文件的话，该值为未知类型 FT_UNKNOWN。

好啦，头文件结束啦，下面看实现，见代码 14-15。

代码 14-15　（project/c14/c/fs/fs.c）

```
…略
229 /* 搜索文件 pathname，若找到则返回其 inode 号，否则返回-1 */
230 static int search_file(const char* pathname, \
struct path_search_record* searched_record) {
231     /* 如果待查找的是根目录，为避免下面无用的查找，
        直接返回已知根目录信息 */
232     if (!strcmp(pathname, "/") || !strcmp(pathname, "/.") ||\
        !strcmp(pathname, "/..")) {
233        searched_record->parent_dir = &root_dir;
234        searched_record->file_type = FT_DIRECTORY;
235        searched_record->searched_path[0] = 0;    // 搜索路径置空
236        return 0;
237     }
238
239     uint32_t path_len = strlen(pathname);
240     /* 保证 pathname 至少是这样的路径/x，且小于最大长度 */
241     ASSERT(pathname[0] == '/' && path_len > 1 &&\
        path_len < MAX_PATH_LEN);
242     char* sub_path = (char*)pathname;
243     struct dir* parent_dir = &root_dir;
244     struct dir_entry dir_e;
245
246     /* 记录路径解析出来的各级名称，如路径"/a/b/c",
247      * 数组 name 每次的值分别是"a","b","c" */
248     char name[MAX_FILE_NAME_LEN] = {0};
249
250     searched_record->parent_dir = parent_dir;
251     searched_record->file_type = FT_UNKNOWN;
252     uint32_t parent_inode_no = 0;  // 父目录的 inode 号
253
254     sub_path = path_parse(sub_path, name);
255     while (name[0]) {       // 若第一个字符就是结束符，结束循环
256        /* 记录查找过的路径，但不能超过 searched_path 的长度 512 字节 */
257        ASSERT(strlen(searched_record->searched_path) < 512);
258
259        /* 记录已存在的父目录 */
260        strcat(searched_record->searched_path, "/");
261        strcat(searched_record->searched_path, name);
262
263        /* 在所给的目录中查找文件 */
264        if (search_dir_entry(cur_part, parent_dir, name, &dir_e)) {
265            memset(name, 0, MAX_FILE_NAME_LEN);
266            /* 若 sub_path 不等于 NULL，也就是未结束时继续拆分路径 */
267            if (sub_path) {
268                sub_path = path_parse(sub_path, name);
269            }
270
271            if (FT_DIRECTORY == dir_e.f_type) {   // 如果被打开的是目录
272                parent_inode_no = parent_dir->inode->i_no;
273                dir_close(parent_dir);
274                parent_dir = dir_open(cur_part, dir_e.i_no); // 更新父目录
275                searched_record->parent_dir = parent_dir;
276                continue;
277            } else if (FT_REGULAR == dir_e.f_type) {   // 若是普通文件
278                searched_record->file_type = FT_REGULAR;
279                return dir_e.i_no;
280            }
281        } else {    //若找不到，则返回-1
282            /* 找不到目录项时，要留着 parent_dir 不要关闭，
283             * 若是创建新文件的话需要在 parent_dir 中创建 */
284            return -1;
285        }
286     }
287
288     /* 执行到此，必然是遍历了完整路径，
```

```
        并且查找的文件或目录只有同名目录存在 */
289     dir_close(searched_record->parent_dir);
290
291     /* 保存被查找目录的直接父目录 */
292     searched_record->parent_dir = dir_open(cur_part, parent_inode_no);
293     searched_record->file_type = FT_DIRECTORY;
294     return dir_e.i_no;
295 }
…略
```

函数 search_file 接受 2 个参数，被检索的文件 pathname 和路径搜索记录指针 searched_record，功能是搜索文件 pathname，若找到则返回其 inode 号，否则返回−1。其中参数 pathname 是全路径，即从根目录开始的路径。参数 searched_record 由主调函数提供，主调函数只关注该结构中的信息，并不关注搜索过程，该结构中的数据由函数 search_file 填充，其中的成员 parent_dir 记录的是待查找目标（文件或目录）的直接父目录，原因是主调函数通常需要获取目标的父目录作为操作对象，比如将来创建文件时，需要知道在哪个目录中创建文件，因此所有调用 search_file 的主调函数记得释放目录 searched_record->parent_dir，避免内存泄漏。下面开始介绍函数实现。

有时候用户输入的路径可能仅是根目录'/'，不包括子路径，比如执行命令“ls/”就是这样的情况。因此在代码第 232～236 行判断，如果待查找的是根目录，为避免后续无用的查找工作，直接在 searched_record 中写入根目录信息后返回。

第 242～244 行用指针 sub_path 指向路径名 pathname，我们后面的处理主要用 sub_path 变量，声明目录指针 parent_dir 指向根目录，我们要从根目录开始往下查找文件，然后声明了目录项 dir_e。

第 248 行声明了数组 name[MAX_FILE_NAME_LEN]，它用来存储路径解析中的各层路径名，您猜到了，name 必然会作为函数 path_parse 的参数。

第 250～252 行也是一些准备工作，其中 parent_inode_no 是父目录 inode 编号，它的作用是备份各层解析出来的路径的父目录的 inode 编号（有点啰嗦，但是无比精确，忽略我吧^_^）。我们从根目录开始解析路径，因此初始化 parent_inode_no 为根目录的 inode 编号 0。

下面开始搜索文件啦。搜索文件的原理是路径解析，也就是把路径按照分隔符'/'拆分，每解析出一层路径名就去目录中确认相应的目录项，与目录项中的 filename 比对，找到后继续路径解析，直到路径解析完成或找不到某个中间目录就返回。

第 254 行执行“sub_path = path_parse(sub_path, name)”开始路径解析，path_parse 返回后，最上层的路径名会存储在 name 中，返回值存入 sub_path，此时的 sub_path 已经剥去了最上层的路径。

第 255 行，使用 while 循环处理各层路径，其判断条件是 name[0]，只要 name[0]不等于字符串结束符'\0'（其 ASCII 码值为 0），就表示 name 不是空字符串，路径解析尚未结束。

循环体中的第 260～261 行，每次解析过的路径都会追加到 searched_record->searched_path 中，searched_path 用于记录已解析的路径，由于是先调用 path_parse 解析路径，再调用 search_dir_entry 去验证路径是否存在，因此 searched_record->searched_path 中的最后一级目录未必存在，其前的所有路径都是存在的。第 260 行的代码在第一次执行时所添加的'/'表示根目录，后续循环中添加的'/'是路径分隔符。

接着在第 264 行调用 search_dir_entry 判断解析出来的上层路径 name 是否在父目录 parent_dir 中存在，如果存在，也就是被找到了，dir_e 中会被录入目录项的信息。接着调用 memset 将 name 清 0，因为后面我们要继续用 name 来存储新的路径。第 267 行判断 sub_path 是否为 null，它是由函数 path_parse 赋值的，该函数总是返回最上层路径之外的子路径。若 sub_path 不为 null，这说明路径解析未完成，还有子路径未拆出来，在第 268 行再次执行“sub_path = path_parse(sub_path, name)”进行下一步的路径解析。

经过前面的 search_dir_entry 调用，dir_e 中已经是目录项的信息了，在第 271 行通过“if(FT_DIRECTORY == dir_e.f_type)”来判断解析出的最上层路径 name 是否为目录，若是目录，就将父目录的 inode 编号赋值给变量 parent_inode_no，此变量用于备份父目录的 inode 编号，它会在最后一级路径为目录的情况下用到，也就是第 292 行代码。接着在第 273 行关闭父目录 parent_dir，打开的目录记得关闭，否则造成内存泄漏。注意，根目录是不可被关闭的，我们在 dir_close 中有特殊处理。接下来把目录 name

打开，重新为 parent_dir 赋值，此处对应的代码是第 274 行的“parent_dir =dir_open(cur_part, dir_e.i_no)”。同时通过第 275 行的代码“searched_record->parent_dir = parent_dir”更新搜索记录中的父目录。如果 name 为普通文件的话，在正常情况下，也就是路径输入正确的情况下，这说明已经把路径从左到右处理完了，因此将搜索记录中的 file_type 更新为 FT_REGULAR，随后返回普通文件 name 的 inode 编号，即 dir_e.i_no。说到此处，细心的朋友可能看出了一些端倪，也许认为这样处理不妥，您的想法我理解，比如拿路径“/a/b”来说，我们从路径上推断 a 是目录，它应该在根目录中。若实际情况是根目录中只有文件 a，并没有目录 a，我们应该返回−1 并给出提示没有目录 a，对吗？我必须得打消您的疑虑。search_file 只负责在文件系统上检索文件，它不负责侦错处理，它是由主调函数调用的，具体的处理方法是由主调函数安排的。也许您又在问，主调函数如何知道 search_file 在搜索过程中发生了什么？您看，search_file 已经把路径处理信息保存在“路径搜索记录”path_search_record 中了，每处理一层路径，都会将其追加到 path_search_record 中的 searched_path 中，因此主调函数可以根据 searched_path 来判断 pathname 是否正确，是否处理完了，待将来我们有 search_file 的应用环境时您就更清楚啦。

若第 264 行的 search_dir_entry 返回 false，也就是未找到 name，程序跳到第 281 行，直接返回−1。注意，未找到 name 时，parent_dir 也不能关闭，因为有可能主调函数会在 parent_dir 目录中创建文件 name，也就是说，主调函数需要知道在哪个目录中创建文件，此时 searched_record->parent_dir 指向父目录，主调函数负责关闭该目录。

程序若能执行到第 289 行，这说明两件事。

（1）路径 pathname 已经被完整地解析过了，各级都存在。

（2）pathname 的最后一层路径不是普通文件，而是目录。

结论是待查找的目标是目录，如“/a/b/c”，c 是目录，不是普通文件。此时 searched_record-> parent_dir 是路径 pathname 中的最后一级目录 c，并不是倒数第二级的父目录 b，我们在任何时候都应该使 searched_record->parent_dir 是被查找目标的直接父目录，也就是说，无论目标是普通文件，还是目录，searched_record->parent_dir 中记录的都应该是目录 b。因此我们需要把 searched_record->parent_dir 重新更新为父目录 b。在重新打开父目录之前，为避免内存溢出，先调用 dir_close 关闭目录 searched_record->parent_dir。接下来是重新打开父目录，我们之前已经将父目录的 inode 编号保存在了变量 parent_inode_no 中，此时是它派上用场的时候，在第 292 行打开父目录并为 searched_record->parent_dir 赋值。然后在下一行更新成员 file_type 为 FT_DIRECTORY，最后返回目录的 inode 编号。

search_file 到这就介绍完了，前期工作铺垫的差不多了，下节咱们该实践了，兄弟们再会。

14.5 创建文件

经过前期大量的准备工作，现在离目标已经很接近了，其实咱们离创建文件只差一步，注意，这里是指创建普通文件，不包括目录，创建目录的工作咱们安排在后续章节。

14.5.1 实现 file_create

为实现文件的创建工作，现在在 file.c 中再增加一个函数 file_create。此函数之所以未在之前介绍 file.c 中时一同讲解，原因是此函数中用到了当时未介绍的函数，不想给大伙儿增加学习的麻烦。

在进行介绍之前，下面讨论下创建文件需要考虑哪些工作。

（1）文件需要 inode 来描述大小、位置等属性，所以创建文件就要创建其 inode。这就涉及到向 inode_bitmap 申请位图来获得 inode 号，因此 inode_bitmap 会被更新，inode_table 数组中的某项也会由新的 inode 填充。

（2）inode->i_sectors 是文件具体存储的扇区地址，这需要向 block_bitmap 申请可用位来获得可用的块（在我们这里，为简化处理，1 块等于 1 扇区），因此 block_bitmap 会被更新，分区的数据区 data_start_lba 以后的某个扇区会被分配。

（3）新增加的文件必然存在于某个目录，所以该目录的 inode->i_size 会增加个目录项的大小。此新增加的文件对应的目录项需要写入该目录的 inode->i_sectors[]中的某个扇区，原有扇区可能已满，所以有可能要申请新扇区来存储目录项。

（4）若其中某步操作失败，需要回滚之前已成功的操作。

（5）inode_bitmap、block_bitmap、新文件的 inode 及文件所在目录的 inode，这些位于内存中已经被改变的数据要同步到硬盘。

该说的都说了，咱们在实践中验证吧，代码走起，见代码 14-16。

代码 14-16　（project/c14/c/fs/file.c）

```
…略
96 /*创建文件，若成功则返回文件描述符，否则返回-1 */
97 int32_t file_create(struct dir* parent_dir, char* filename, uint8_t flag) {
98     /* 后续操作的公共缓冲区 */
99     void* io_buf = sys_malloc(1024);
100    if (io_buf == NULL) {
101        printk("in file_creat: sys_malloc for io_buf failed\n");
102        return -1;
103    }
104
105    uint8_t rollback_step = 0;   // 用于操作失败时回滚各资源状态
106
107    /* 为新文件分配 inode */
108    int32_t inode_no = inode_bitmap_alloc(cur_part);
109    if (inode_no == -1) {
110        printk("in file_creat: allocate inode failed\n");
111        return -1;
112    }
113
114 /* 此 inode 要从堆中申请内存，不可生成局部变量（函数退出时会释放），
115  * 因为 file_table 数组中的文件描述符的 inode 指针要指向它 */
116    struct inode* new_file_inode =\
       (struct inode*)sys_malloc(sizeof(struct inode));
117    if (new_file_inode == NULL) {
118        printk("file_create: sys_malloc for inode failded\n");
119        rollback_step = 1;
120        goto rollback;
121    }
122    inode_init(inode_no, new_file_inode);     // 初始化 i 结点
123
124 /* 返回的是 file_table 数组的下标 */
125    int fd_idx = get_free_slot_in_global();
126    if (fd_idx == -1) {
127        printk("exceed max open files\n");
128        rollback_step = 2;
129        goto rollback;
130    }
131
132    file_table[fd_idx].fd_inode = new_file_inode;
133    file_table[fd_idx].fd_pos = 0;
134    file_table[fd_idx].fd_flag = flag;
135    file_table[fd_idx].fd_inode->write_deny = false;
136
137    struct dir_entry new_dir_entry;
138    memset(&new_dir_entry, 0, sizeof(struct dir_entry));
139
140    create_dir_entry(filename, inode_no, FT_REGULAR, &new_dir_entry);
       // create_dir_entry 只是内存操作不出意外，不会返回失败
141
142 /* 同步内存数据到硬盘 */
143    /* a 在目录 parent_dir 下安装目录项 new_dir_entry,
       写入硬盘后返回 true，否则 false */
144    if (!sync_dir_entry(parent_dir, &new_dir_entry, io_buf)) {
145        printk("sync dir_entry to disk failed\n");
146        rollback_step = 3;
147        goto rollback;
148    }
149
```

```
150     memset(io_buf, 0, 1024);
151     /* b 将父目录 i 结点的内容同步到硬盘 */
152     inode_sync(cur_part, parent_dir->inode, io_buf);
153
154     memset(io_buf, 0, 1024);
155     /* c 将新创建文件的 i 结点内容同步到硬盘 */
156     inode_sync(cur_part, new_file_inode, io_buf);
157
158     /* d 将 inode_bitmap 位图同步到硬盘 */
159     bitmap_sync(cur_part, inode_no, INODE_BITMAP);
160
161     /* e 将创建的文件 i 结点添加到 open_inodes 链表 */
162     list_push(&cur_part->open_inodes, &new_file_inode->inode_tag);
163     new_file_inode->i_open_cnts = 1;
164
165     sys_free(io_buf);
166     return pcb_fd_install(fd_idx);
167
168 /*创建文件需要创建相关的多个资源,
    若某步失败则会执行到下面的回滚步骤 */
169 rollback:
170     switch (rollback_step) {
171        case 3:
172           /* 失败时，将 file_table 中的相应位清空 */
173           memset(&file_table[fd_idx], 0, sizeof(struct file));
174        case 2:
175           sys_free(new_file_inode);
176        case 1:
177           /* 如果新文件的 i 结点创建失败,
              之前位图中分配的 inode_no 也要恢复 */
178           bitmap_set(&cur_part->inode_bitmap, inode_no, 0);
179           break;
180     }
181     sys_free(io_buf);
182     return -1;
183 }
```

函数 file_create 接受 3 个参数，父目录 partent_dir、文件名 filename、创建标识 flag，功能是在目录 parent_dir 中以模式 flag 去创建普通文件 filename，若成功则返回文件描述符，即 pcb->fd_table 中的下标，否则返回−1。大伙儿最好是结合图 14-16，这样便于理解 file_create 在做什么。

file_create 是基于之前介绍的基础函数，如函数 inode_sync 和 sync_dir_entry。这两个是往硬盘上写数据的函数，其原理是需要先把原扇区的数据读到内存，在内存中将数据变更后再写入硬盘扇区，所以用于变更数据的内存缓冲区是不可少的。往硬盘上同步数据的操作往往是诸多步骤中的最后一步，如果在这类函数内部申请内存作为缓冲区，万一内存不足，则往硬盘上同步数据就会失败，那之前所做的所有工作都会白费，所以在创建文件之初就应该把缓冲区准备好，如果申请内存失败了也不会多做无用功，也避免了最后无法同步到硬盘而造成数据不一致、回滚操作过多的情况发生。一般情况下硬盘操作都是一次读写一个扇区，考虑到有数据会跨扇区的情况，故申请 2 个扇区大小的缓冲区，因此在函数开头就先申请了 1024 字节的缓冲区 io_buf。

现在再说一下回滚，回滚就是资源变更后，将资源恢复到未修改前的状态。文件系统是一套资源管理的方法，每变动一种数据就会涉及到多种资源的“联动”，这里所说的“联动”意指一个事物的状态改变后，周边事物也要一同跟着变动的连锁反应。按理说相关资源的联动应该是一个事务，具有原子性，即要么都成功变更，要么都不变，万一在哪个步骤中失败了，其前所做的变动都要回滚到之前的状态。这就像超市里卖货一样，卖出去一件商品后，营业额账本上要增加一笔流水，库存中要减少一件商品，当该商品被顾客退回的时候，账本上要减去这笔流水，并且库存中要增加一件商品。

创建文件包括多个修改资源的步骤，我们创建新文件的顺序是：创建文件 i 结点->文件描述符 fd ->目录项。这种“从后往前”创建步骤的好处是每一步创建失败时回滚操作少。不过随着步骤的递增，失败时回滚的步骤也将递增。这里第 105 行定义了变量 rollback_step 用于记录回滚步骤。真正用于回滚的代码是在第 169 行的标签 rollback 处，大家移步过去看看，那里有 3 部分的回滚操作，只列出了有可能会失败的操作，从上到下依次是 case 3、case2、case1，各 case 之间没有 break，它们是一种累加的回滚，因此 case3

执行的回滚操作最多，case1 最少，需要回滚时，只要将 rollback_step 置为相应的 case 就好了。

第 108 行调用 inode_bitmap_alloc 为新文件分配 inode，第 116 行为新文件的 inode——new_file_inode 申请内存，如果内存申请失败，rollback_step 置为 1，程序跳转到 rollback 处，也就是第 169 行，根据 rollback_step 的值，会执行分支 case 1，即执行第 178 行的 bitmap_set 回滚位图状态。如果内存分配成功的话，执行 inode_init 初始化 new_file_inode。

接下来调用 get_free_slot_in_global 从 file_talbe 中获取空闲文件结构的下标，写入变量 fd_idx 中。如果 file_table 中没有空闲位则返回−1，于是将 rollback_step 置为 2，依然是通过 goto 语句跳转到 rollback 处，执行 case2 处的代码，执行 sys_free(new_file_inode)释放 new_file_inode 的内存，然后执行 case1 恢复 inode 位图。

第 132～135 行初始化文件表中的文件结构，第 137～138 行为文件创建新目录项 new_dir_ entry，并将其清 0，第 140 行调用 create_dir_entry 用 filename、inode_no 和 FT_REGULAR 填充 new_dir_entry。准备好目录项后，接着在第 144 行通过函数 sync_dir_entry (parent_dir, &new_dir_entry, io_buf)将其写入到父目录 parent_dir 中，如果失败，rollback_step 置为 3，执行回滚。

sync_dir_entry 会改变父目录 inode 中的信息，因此在第 152 行调用函数 inode_sync 将父目录 inode 同步到硬盘。接着在第 156～159 行分别将新文件的 inode 同步到硬盘，将 inode_bitmap 位图同步到硬盘。第 162 行将新文件的 inode 添加到 inode 列表，也就是 cur_part->open_inodes，随后在其 i_open_cnts 置为 1。以上几个同步操作一般不会出问题，因此这并未有相应的回滚。硬盘操作到这就结束了，在第 165 行将 io_buf 释放，然后在第 166 行调用 pcb_fd_install(fd_idx)，在数组 pcb->fd_table 中找个空闲位安装 fd_idx，若成功则返回空闲位的下标，若失败则返回−1，用 return 将其返回值返回。

好啦，file_create 就完成了，下节咱们要用它创建文件了。

14.5.2　实现 sys_open

终于要实现创建文件啦，等了好久终于等到今天。

本节要实现的函数是 sys_open，它是 open 函数的内核级实现，有了它之后将来实现 open 系统调用就很容易了。不知道是否有同学会觉得奇怪，哎？创建文件不是应该用 creat 函数吗？其原型是“int creat(const char *pathname, mode_t mode)”啊，是啊，但 open 函数功能更多，它提供了很多 flag，因此除了能打开文件外，还能创建新文件，以下面的形式调用 open 就相当于 creat。

open(const char * pathname, (O_CREAT|O_WRONLY|O_TRUNC));

这里不打算单独实现 creat 函数，open 函数功能更通用，还是以它为主吧，了解了原理之后，creat 函数实现也很容易的。好啦，上代码，见代码 14-17。

代码 14-17　（project/c14/c/fs/fs.c）

```
…略
297 /* 打开或创建文件成功后，返回文件描述符，否则返回-1 */
298 int32_t sys_open(const char* pathname, uint8_t flags) {
299    /* 对目录要用 dir_open，这里只有 open 文件 */
300     if (pathname[strlen(pathname) - 1] == '/') {
301          printk("can`t open a directory %s\n",pathname);
302          return -1;
303     }
304     ASSERT(flag <=7);
305     int32_t fd = -1;       // 默认为找不到
306
307     struct path_search_record searched_record;
308     memset(&searched_record, 0, sizeof(struct path_search_record));
309
310     /* 记录目录深度，帮助判断中间某个目录不存在的情况 */
311     uint32_t pathname_depth = path_depth_cnt((char*)pathname);
312
313     /* 先检查文件是否存在 */
314     int inode_no = search_file(pathname, &searched_record);
315     bool found = inode_no != -1 ? true : false;
316
317     if (searched_record.file_type == FT_DIRECTORY) {
```

```
318         printk("can`t open a direcotry with open(), use opendir() to instead\n");
319         dir_close(searched_record.parent_dir);
320         return -1;
321     }
322
323     uint32_t path_searched_depth = \
        path_depth_cnt(searched_record.searched_path);
324
325     /* 先判断是否把 pathname 的各层目录都访问到了,
        即是否在某个中间目录就失败了 */
326     if (pathname_depth != path_searched_depth) {
        // 说明并没有访问到全部的路径，某个中间目录是不存在的
327         printk("cannot access %s: Not a directory, subpath %s is't exist\n", \
328                 pathname, searched_record.searched_path);
329         dir_close(searched_record.parent_dir);
330         return -1;
331     }
332
333     /* 若是在最后一个路径上没找到，并且并不是要创建文件，直接返回-1 */
334     if (!found && !(flags & O_CREAT)) {
335         printk("in path %s, file %s is`t exist\n", \
336                 searched_record.searched_path, \
337                 (strrchr(searched_record.searched_path, '/') + 1));
338         dir_close(searched_record.parent_dir);
339         return -1;
340     } else if (found && flags & O_CREAT) {  // 若要创建的文件已存在
341         printk("%s has already exist!\n", pathname);
342         dir_close(searched_record.parent_dir);
343         return -1;
344     }
345
346     switch (flags& O_CREAT) {
347         case O_CREAT:
348             printk("creating file\n");
349             fd = file_create(searched_record.parent_dir, \
            (strrchr(pathname, '/') + 1), flags);
350             dir_close(searched_record.parent_dir);
351         // 其余为打开文件
352     }
353
354     /* 此 fd 是指任务 pcb->fd_table 数组中的元素下标，
355      * 并不是指全局 file_table 中的下标 */
356     return fd;
357 }
```

sys_open 函数接受 2 个参数，pathname 是待打开的文件，其为绝对路径，flags 是打开标识，其值便是之前在 fs.h 头文件中提前放入的 enum oflags。函数功能是打开或创建文件成功后，返回文件描述符，即 pcb 中 fd_table 中的下标，否则返回−1。

目录以字符'/'结尾，如“/a/”，a 便是指目录，sys_open 只支持文件打开，不支持目录打开，因此程序开头判断 pathname 是否为目录，这里是对 pathname 的最后一个字符判断，即若 pathname[strlen(pathname) - 1] 等于'/'，就表示 pathname 为目录，打印提示信息并返回−1。

第 304 行的“ASSERT(flag <=7)”是限制 flags 的值在 O_RDONLY| O_WRONLY| O_RDWR| O_CREAT 之内。第 305 声明了变量 fd，为其初始化为−1，即默认找不到文件。

第 307 行生成了路径搜索记录变量 searched_record，path_search_record 用来记录文件查找时所遍历过的目录，它会作为参数传给函数 search_file，其值由函数 search_file 填充。当查找失败时，主调函数可以根据此结构了解在哪层子目录下失败了。若是创建文件，便可直接获得所创建文件的父目录，即在哪个目录下创建文件。path_search_record 中一个很重要的成员是 searched_path，它用来记录所处理过的路径，可以通过该路径的长度来判断查找是否成功。举个例子，若查找目标文件 c，它的绝对路径是“/a/b/c”，查找时若发现 b 目录不存在，存入 path_search_ record.searched_path 的内容便是“/a/b”，若按照此路径找到了 c，path_search_record.searched_path 的值便是完整路径“/a/b/c”。

第 308 行将 searched_record 清 0。由于 searched_record 位于栈中，栈中数据并不会自动清 0，这非常

容易出问题。尤其是在连续无间隔调用 sys_open 打开不同文件的时候，如果有的文件不存在，有的中间路径缺失，有的成功打开，search_file 函数会在 searched_record 中记录不同的状态，下一次调用 sys_open，searched_recored 还是指向栈中相同的位置，栈中数据不会自动清 0，所以数据还是上一次调用中的结果，这影响 sys_open 中相关代码的判断结果，所以在使用前一定要先清 0。

第 311 行通过 path_depth_cnt 计算 pathname 的深度，深度值写入变量 pathname_depth，计算目录深度的目的是帮助判断某个目录不存在的情况，一会儿咱们就知道了。

无论是打开文件，还是创建文件，都要先判断文件是否已存在，在第 314 行调用 search_file 搜索文件 pathname，搜索结果存入 searched_record。第 315 行为 bool 变量 found 赋值。

第 317 行判断 pathname 的判断，若为目录，在第 318 行打印提示信息，接着调用 dir_close 关闭目录 searched_record.parent_dir，并返回−1。前面咱们说过，search_file 返回的 path_search_ record.parent_dir 由主调函数负责关闭，原因是主调函数有可能会用到此目录，也许会在该目录下创建文件，后面第 349 行的 file_create 便是此运用场合。

第 323 行调用 path_depth_cnt 计算 searched_record.searched_path 的路径深度，值写入变量 path_searched_depth 中。接着在第 326 行用原路径 pathname 的深度值 pathname_depth 与 path_searched_depth 对比，若不相等，说明查找文件的过程中并没有访问到全部的路径，某个中间目录或最后目标不存在，总之检索失败，于是在第 327 行输出报错信息，告诉用户哪个子目录不存在，便于用户知道在哪里输错了。接着关闭目录，返回−1。

当 flags 包含 O_CREAT 时，open 函数可以创建文件。在第 334 行判断，若目标文件未找到，并且 flags 不包含 O_CREAT，这说明想打开的文件不存在，并不是想创建文件，因此在下一行输出报错信息，然后关闭目录 searched_record.parent_dir 并返回−1。

第 340 行判断，若找到了文件并且 flags 包含 O_CREAT，这说明想创建的文件名已存在，相同目录下不允许同名文件存在，因此输出报错并关闭目录，返回−1。

经过重重考验之后，终于到了创建文件的时刻，第 346 的 switch 结构根据 flags 中是否包括 O_CREAT 的情况暂时只建立了 O_CREAT 分支，即目前只支持 sys_open(“xxx”,O_CREAT|O_XXX)的用法。创建文件是用 file_create 实现的，因此在第 349 行调用 file_create 创建文件，返回文件描述符存入变量 fd，完成后关闭目录 searched_record.parent_dir，并返回 fd，至此创建文件完成。

好啦，sys_open 完成了，还有一点小尾巴没有介绍，咱们还没有打开根目录并初始化文件表呢，我们把这部分代码放在 filesys_init 的末尾，如代码 14-18 所示。

代码 14-18　（project/c14/c/fs/fs.c）

```
…略
364 void filesys_init() {
…略
415    /* 确定默认操作的分区 */
416    char default_part[8] = "sdb1";
417    /* 挂载分区 */
418    list_traversal(&partition_list, mount_partition, (int)default_part);
419
420    /* 将当前分区的根目录打开 */
421    open_root_dir(cur_part);
422
423    /* 初始化文件表 */
424    uint32_t fd_idx = 0;
425    while (fd_idx < MAX_FILE_OPEN) {
426       file_table[fd_idx++].fd_inode = NULL;
427    }
428 }
```

代码第 421 行以下的内容是本节新增的内容，包括调用 open_root_dir 打开根目录和初始化文件表，使文件结构中的 fd_inode 置为 NULL，表示该文件结构为空位，可分配。

到这本节也就结束了，下节咱们功能验证。

14.5.3　在文件系统上创建第 1 个文件

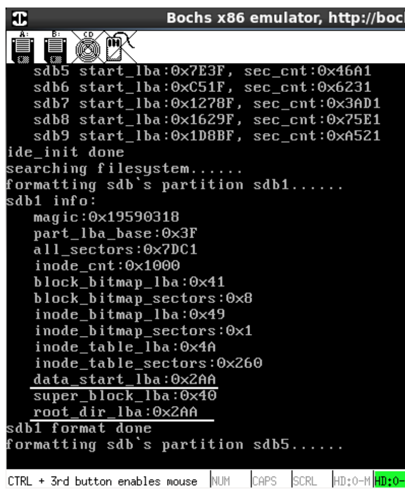
本节是功能测试，检测 sys_open 创建文件的功能，在 main.c 中加入其调用，见代码 14-19。

代码 14-19　（project/c14/c/kernel/main.c）

```
…略
18 int main(void) {
19      put_str("I am kernel\n");
20      init_all();
21      process_execute(u_prog_a, "u_prog_a");
22      process_execute(u_prog_b, "u_prog_b");
23      thread_start("k_thread_a", 31, k_thread_a, "I am thread_a");
24      thread_start("k_thread_b", 31, k_thread_b, "I am thread_b");
25      sys_open("/file1", O_CREAT);
26      while(1);
27      return 0;
28 }
…略
```

代码就这些，在第 25 行加入了“sys_open("/file1", O_CREAT)”，在根目录下创建文件 file1。目前我们只有根目录这一个目录，因此只能把文件创建在它之下，待以后完成创建目录的功能后咱们再自由发挥吧。

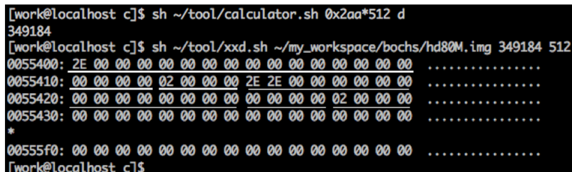
在创建文件之前，咱们先看下文件被创建在哪里，一会方便咱们验证，如图 14-18 所示。



▲图 14-18　sdb1 数据区地址

图中用下画线标识的是数据区起始扇区地址和根目录起始扇区地址，它们应该是相同的，毕竟根目录也属于数据区的范围，只是根目录占据的空间是数据区最开始的扇区。根目录扇区地址是十六进制 0x2aa，我们将其换算成十进制的字节表示后，再通过 xxd.sh 脚本查看 hd80M.img，过程如图 14-19 所示。

您看，0x2aa 是扇区，将其乘以 512 后，十进制结果是 349184，接着调用 xxd.sh 查看 hd80M.img 偏移 349184 字节处的 512 字节的内容。目录中的内容是目录项，每个目录项包括三部分的内容：16 字节的文件名 filename，4 字节的 inode 编号 i_no，4 字节的文件类型 f_type，即目录项总大小是 24 字节。如图所示，目前根目录中就两个目录项，分别用粗线和细线分别标出。



▲图 14-19　根目录中的内容

各个目录中都有'.'和'..'，它们分别表示当前目录和上级直接父目录。根目录是所有目录的父目录，没有上级了，因此对根目录中的'..'做了特殊处理，使其指向根目录自身。

第一个目录项是偏移字节 0x55400 字节处的连续 24 字节，其中低 16 字节中除最低处是 0x2e，其他都是 0，这 16 字节是目录项中的文件名 filename，0x2e 是字符'.'的 ASCII 码。第二段粗线是中间的 4 字节，这是 inode 编号 i_no，其值为 0，这表示'.'的 inode 编号就是根目录的 inode 编号 0。第三段粗线是目录项的最后 4 字节，这是文件类型 f_type，其值为 2，表示'.'是目录类型 FT_DIRECTORY，以上就是我们熟悉的目录中的文件 “.” 的目录项。后面细线标出的是第二个目录项，您也许猜到了，这是”..”的目录项内容，前 16 字节表示 filename 是”..”，第二段细线表示 i_no，其值为 0，这说明”..”是指向根目录的 inode 编号 0，最后一段细线表示”..”是目录。另外说明一下，在图 14-19 中右边的显示区是用来显示字符的，如果左边的字节无法按照 ASCII 码解析成可见字符，就会用'.'来代替，每个'.'都与左边的 1 个字节对应，因此每行共 16 个'.'。而实际上我们的两个目录名'.'和'..'正巧和表示不可见字符的'.'重合了，因此似乎在右边显示区中看不到文件名，其实是有的，一会后面我们创建 file1 后会验证。

目前根目录中就只有目录'.'和'..'，这说明根目录下尚未创建文件。下面我们运行程序，用“sys_open("/file1", O_CREAT)”在根目录下创建文件 file1，过程如图 14-20 所示。

为了使创建文件不那么低调，这里在 sys_open 函数中打印了“creating file”，至少我们知道程序在工作。下面再看下根目录的内容，如图 14-21 所示。

图中偏移 0x55430 的地方是第三个文件 file1 的目录项，我们已经用最粗的线标出了，前两类线是'.'、'..'和目录项。在右边的显示区中已经显示了字符串 file1，它分别对应 ASCII 码 0x66，0x69，0x6c，0x65 和 0x31。以 0x55440 开始的 4 字节是 i_no，其值为 1，这说明 file1 的 inode 编号是 1，是紧跟着根目录之后的 inode，最后一段是 f_type，其值为 1，说明是普通文件 FT_REGULAR。

由于此时根目录中已经存在了 file1，下面我们再运行一次程序看看，结果如图 14-22 所示。

```
Bochs x86 emulator, http://bochs.sourcef
     SECTORS: 163296
     CAPACITY: 79MB

   all partition info
   sdb1 start_lba:0x3F, sec_cnt:0x7DC1
   sdb5 start_lba:0x7E3F, sec_cnt:0x46A1
   sdb6 start_lba:0xC51F, sec_cnt:0x6231
   sdb7 start_lba:0x1278F, sec_cnt:0x3AD1
   sdb8 start_lba:0x1629F, sec_cnt:0x75E1
   sdb9 start_lba:0x1D8BF, sec_cnt:0xA521
ide_init done
searching filesystem......
sdb1 has filesystem
sdb5 has filesystem
sdb6 has filesystem
sdb7 has filesystem
sdb8 has filesystem
sdb9 has filesystem
mount sdb1 done!
 prog_a malloc addr:0x804800C,0x804810C,0x804820C
 prog_b malloc addr:0x804800C,0x804810C,0x804820C
 thread_a malloc addr:0xC013900C,C013910C,C013920C
 thread_b malloc addr:0xC013930C,C013940C,C013950C
creating file
CTRL + 3rd button enables mouse  NUM  CAPS  SCRL  HD:0-M HD:0-S
```

▲图 14-20　创建文件 file1

```
[work@localhost c]$ sh ~/tool/xxd.sh ~/my_workspace/bochs/hd80M.img 349184 512
0055400: 2E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0055410: 00 00 00 00 02 00 00 00 2E 2E 00 00 00 00 00 00  ................
0055420: 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00  ................
0055430: 66 69 6C 65 31 00 00 00 00 00 00 00 00 00 00 00  file1...........
0055440: 01 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00  ................
0055450: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00555f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[work@localhost c]$
```

▲图 14-21　根目录中产生了新文件

```
Bochs x86 emulator, http://bochs.source
     SECTORS: 163296
     CAPACITY: 79MB

   all partition info
   sdb1 start_lba:0x3F, sec_cnt:0x7DC1
   sdb5 start_lba:0x7E3F, sec_cnt:0x46A1
   sdb6 start_lba:0xC51F, sec_cnt:0x6231
   sdb7 start_lba:0x1278F, sec_cnt:0x3AD1
   sdb8 start_lba:0x1629F, sec_cnt:0x75E1
   sdb9 start_lba:0x1D8BF, sec_cnt:0xA521
ide_init done
searching filesystem......
sdb1 has filesystem
sdb5 has filesystem
sdb6 has filesystem
sdb7 has filesystem
sdb8 has filesystem
sdb9 has filesystem
mount sdb1 done!
 prog_a malloc addr:0x804800C,0x804810C,0x804820C
 prog_b malloc addr:0x804800C,0x804810C,0x804820C
 thread_a malloc addr:0xC013900C,C013910C,C013920C
 thread_b malloc addr:0xC013930C,C013940C,C013950C
/file1 has already exist!
CTRL + 3rd button enables mouse  NUM  CAPS  SCRL  HD:0-M HD:0-S
```

▲图 14-22　无法创建同名的文件

图中最下行已经打印了“/file1 has already exist!”，这说明根目录下已存在 file1，同一目录下无法创建同名文件，功能符合预期。

好啦，目前检测暂时通过了，本节到这也就结束了，兄弟们休息下，下节再战。

14.6 文件的打开与关闭

因为有关读写的函数都要用文件描述符作为参数，所以文件的读写操作通常是在文件被打开返回描述符之后。尽管上节中我们通过 sys_open 实现了文件创建的功能，但这远远不够，除了我们要改进 sys_open 使其支持更多的功能之外，还要单独编写关闭文件的代码，出发喽。

14.6.1 文件的打开

现在我们已经在根目录下创建了文件 file1，接下来可以编写文件打开的代码了，打开文件的核心操作是 file_open 完成的，它定义在 file.c 中。不耽误各位客官的时间了，上菜，见代码 14-20。

代码 14-20 （project/c14/d/fs/file.c）

```
…略
185 /*打开编号为 inode_no 的 inode 对应的文件，
    若成功则返回文件描述符，否则返回-1 */
186 int32_t file_open(uint32_t inode_no, uint8_t flag) {
187     int fd_idx = get_free_slot_in_global();
188     if (fd_idx == -1) {
189         printk("exceed max open files\n");
190         return -1;
191     }
192     file_table[fd_idx].fd_inode = inode_open(cur_part, inode_no);
193     file_table[fd_idx].fd_pos = 0;
        // 每次打开文件，要将 fd_pos 还原为 0，即让文件内的指针指向开头
194     file_table[fd_idx].fd_flag = flag;
195     bool* write_deny = &file_table[fd_idx].fd_inode->write_deny;
196
197     if (flag & O_WRONLY || flag & O_RDWR) {
        // 只要是关于写文件，判断是否有其他进程正写此文件
198             // 若是读文件，不考虑 write_deny
199     /* 以下进入临界区前先关中断 */
200        enum intr_status old_status = intr_disable();
201        if (!(*write_deny)) {    // 若当前没有其他进程写该文件，将其占用
202             *write_deny = true; // 置为 true，避免多个进程同时写此文件
203             intr_set_status(old_status);     // 恢复中断
204        } else {          // 直接失败返回
205             intr_set_status(old_status);
206             printk("file can’t be write now, try again later\n");
207             return -1;
208        }
209     }  // 若是读文件或创建文件，不用理会 write_deny，保持默认
210     return pcb_fd_install(fd_idx);
211 }
212
```

file_open 接受 2 个参数，inode 编号 inode_no 和打开标识 flag，函数功能是打开编号为 inode_no 的 inode 对应的文件，若成功则返回文件描述符，否则返回−1。

函数开头调用 get_free_slot_in_global 从文件表 file_table 中获取空位的下标，接着在第 192～194 行初始化。

第 195 使变量 write_deny 指向该 inode 的 write_deny 位，下面进行判断和处理。第 197 行，如果此次以写文件的方式打开文件，也就是 flag 中包含 O_WRONLY（只写）或 O_RDWR（读和写），为了避免多个任务同时写该文件而引起相互覆盖的混乱，第 201 行对指针 write_deny 取值，判断是否为 true，检查是否已有别的任务正在写该文件，如果为 false，在第 202 行将其置为 true，表示本任务要对其执行写操作，否则就简单处理，输出提示信息“file can`t be write now, try again later”，也就是目前文件不能写入，一会再试。然后返回−1。

如果 flag 是 O_RDONLY 读文件或 O_CREAT 创建文件，就不用理会 write_deny 的值了，我们允许写文件时对其执行读操作。

最后第 210 行将 fd_idx 安装到 fd_table 并返回结果，若成功则返回文件描述符，否则返回−1。

下面我们改进 sys_open，见代码 14-21。

代码 14-21 （project/c14/d/fs/fs.c）

```
…略
297     /* 打开或创建文件成功后，返回 pcb 中 fd_table 中的下标，否则返回-1 */
298     int32_t sys_open(const char* pathname, uint8_t flag) {
…略
346     switch (flags & O_CREAT) {
347        case O_CREAT:
348           printk("creating file\n");
349           fd = file_create(searched_record.parent_dir, \
             (strrchr(pathname, '/') + 1), flag);
350           dir_close(searched_record.parent_dir);
351           break;
352        default:
353     /* 其余情况均为打开已存在文件
354      * O_RDONLY,O_WRONLY,O_RDWR */
355           fd = file_open(inode_no, flags);
```

```
356     }
…略
```

这里只修改对 flag 判断的 switch 结构，在第 352～355 行增加了其他标识的支持，即 O_RDONLY、O_WRONLY 和 O_RDWR 默认都由函数 file_open 处理。

好啦，有关文件打开的部分就介绍完了。

14.6.2　文件的关闭

文件有打开就得有关闭，Linux 中文件关闭是 close 函数，当然还有其他函数，我们只打算实现类似 close 函数的功能。

close 函数原型是“int close(int fd)”，关闭成功则返回 0，否则返回−1。咱们的实现非常简单，关闭文件的核心操作是 file_close，见代码 14-22。

代码 14-22　（project/c14/d/fs/file.c）

```
…略
213     /* 关闭文件 */
214    int32_t file_close(struct file* file) {
215        if (file == NULL) {
216            return -1;
217     }
218     file->fd_inode->write_deny = false;
219     inode_close(file->fd_inode);
220     file->fd_inode = NULL;   // 使文件结构可用
221     return 0;
222 }
…略
```

file_close 接受 1 个参数，文件 file，功能是关闭文件，成功则返回 0，否则返回−1。

为了兼容 close 的接口，咱们的实现比较简单，file_close 唯一失败返回的情况就是 file 为 null。其余的代码就是恢复 inode 的状态，不多说了，下面看 sys_close 的实现，见代码 14-23。

代码 14-23　（project/c14/d/fs/fs.c）

```
…略
363 /* 将文件描述符转化为文件表的下标 */
364 static uint32_t fd_local2global(uint32_t local_fd) {
365     struct task_struct* cur = running_thread();
366     int32_t global_fd = cur->fd_table[local_fd];
367     ASSERT(global_fd >= 0 && global_fd < MAX_FILE_OPEN);
368     return (uint32_t)global_fd;
369 }
370
371     /* 关闭文件描述符 fd 指向的文件，成功返回 0，否则返回-1 */
372     int32_t sys_close(int32_t fd) {
373     int32_t ret = -1;   // 返回值默认为-1,即失败
374     if (fd > 2) {
375         uint32_t _fd = fd_local2global(fd);
376         ret = file_close(&file_table[_fd]);
377         running_thread()->fd_table[fd] = -1; // 使该文件描述符位可用
378     }
379     return ret;
380 }
…略
```

这里列出了两个函数，fd_local2global 和 sys_close。

fd_local2global 是 sys_close 所依赖的函数，它接受 1 个参数，文件描述符 local_fd，功能是将文件描述符转化为文件表的下标。原理就是将 local_fd 作为下标代入数组 fd_table，fd_table [local_fd]的值便是文件表的下标。

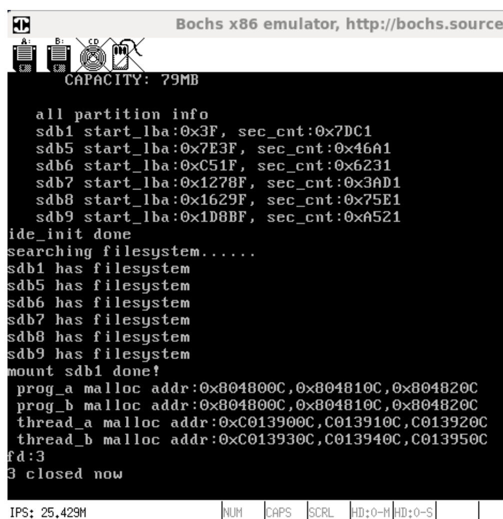
函数 sys_close 接受 1 个参数，文件描述符 fd，功能是关闭文件描述符 fd 指向的文件，成功返回 0，否则返回−1。第 375 行，通过“fd_local2global(fd)”获取 file_table 的下标，存入变量_fd，然后用_fd 索引文件表中的相应文件结构，在下一行把文件结构的指针作为参数调用 file_close 关闭文件。然后把本地文件描述符置为−1，使其为空可分配。

关闭文件的代码就这样精简，没啦。下面测试下文件打开与关闭的功能，还是老样子，在 main.c 中添加测试代码，如代码 14-24 所示。

代码 14-24 （project/c14/d/kernel/main.c）

```
…略
18 int main(void) {
19      put_str("I am kernel\n");
20      init_all();
21      process_execute(u_prog_a, "u_prog_a");
22      process_execute(u_prog_b, "u_prog_b");
23      thread_start("k_thread_a", 31, k_thread_a, "I am thread_a");
24      thread_start("k_thread_b", 31, k_thread_b, "I am thread_b");
25
26      uint32_t fd = sys_open("/file1", O_RDONLY);
27      printf("fd:%d\n", fd);
28      sys_close(fd);
29      printf("%d closed now\n", fd);
30      while(1);
31      return 0;
32 }
…略
```

第 26～29 行是测试代码，第 26 行将“/file1”打开，输出返回的文件描述符后，接着在第 28 行将其关闭，我们看下运行结果吧，如图 14-23 所示。



▲图 14-23 文件打开与关闭

图中最后两行输出了“fd:3”和“3 closed now”，初步符合预期，因此本节画上了句号。

14.7 实现文件写入

本节要实现的 sys_write 是系统调用 write 的内核实现，咱们之前的 write 是个简易版，它是为了临时完成输出打印的功能，不支持文件描述符。如今要让 write 支持文件描述符的话，还要修改下周边与此系统调用相关的内容。

14.7.1 实现 file_write

想必您已经了解了我们实现文件系统功能的“套路”，基本上都是在 file.c 中添加“file_xxx”的核心功能，然后在 fs.c 中添加其外壳，好吧，走起，先实现 file_write，代码很长，由于这只是一个函数的内容，未拆分成多个部分，见代码 14-25。

代码 14-25　（project/c14/e/fs/file.c）

```
...略
224 /* 把 buf 中的 count 个字节写入 file，
成功则返回写入的字节数，失败则返回-1 */
225 int32_t file_write(struct file* file, const void*buf, uint32_t count) {
226     if ((file->fd_inode->i_size + count) > (BLOCK_SIZE * 140))   {
        // 文件目前最大只支持 512*140=71680 字节
227         printk("exceed max file_size 71680 bytes, write file failed\n");
228         return -1;
229     }
230     uint8_t* io_buf = sys_malloc(512);
231     if (io_buf == NULL) {
232         printk("file_write: sys_malloc for io_buf failed\n");
233         return -1;
234     }
235     uint32_t* all_blocks = (uint32_t*)sys_malloc(BLOCK_SIZE + 48);
          // 用来记录文件所有的块地址
236     if (all_blocks == NULL) {
237         printk("file_write: sys_malloc for all_blocks failed\n");
238         return -1;
239     }
240
241     const uint8_t* src = buf;     // 用 src 指向 buf 中待写入的数据
242     uint32_t bytes_written = 0;      // 用来记录已写入数据大小
243     uint32_t size_left = count;      // 用来记录未写入数据大小
244     int32_t block_lba = -1;         // 块地址
245     uint32_t block_bitmap_idx = 0;
        // 用来记录 block 对应于 block_bitmap 中的索引，作为参数传给 bitmap_sync

246     uint32_t sec_idx;         // 用来索引扇区
247     uint32_t sec_lba;         // 扇区地址
248     uint32_t sec_off_bytes;    // 扇区内字节偏移量
249     uint32_t sec_left_bytes;   // 扇区内剩余字节量
250     uint32_t chunk_size;       // 每次写入硬盘的数据块大小
251     int32_t indirect_block_table;  // 用来获取一级间接表地址
252     uint32_t block_idx;         // 块索引
253
254     /* 判断文件是否是第一次写，如果是，先为其分配一个块 */
255     if (file->fd_inode->i_sectors[0] == 0) {
256         block_lba = block_bitmap_alloc(cur_part);
257         if (block_lba == -1) {
258             printk("file_write: block_bitmap_alloc failed\n");
259             return -1;
260         }
261         file->fd_inode->i_sectors[0] = block_lba;
262
263         /* 每分配一个块就将位图同步到硬盘 */
264         block_bitmap_idx = block_lba - cur_part->sb->data_start_lba;
265         ASSERT(block_bitmap_idx != 0);
266         bitmap_sync(cur_part, block_bitmap_idx, BLOCK_BITMAP);
267     }
268
269     /* 写入 count 个字节前，该文件已经占用的块数 */
270     uint32_t file_has_used_blocks = file->fd_inode->i_size / BLOCK_SIZE + 1;
271
272     /* 存储 count 字节后该文件将占用的块数 */
273     uint32_t file_will_use_blocks =\
        (file->fd_inode->i_size + count) / BLOCK_SIZE + 1;
274     ASSERT(file_will_use_blocks <= 140);
275
276     /* 通过此增量判断是否需要分配扇区，如增量为 0，表示原扇区够用 */
277     uint32_t add_blocks = file_will_use_blocks - file_has_used_blocks;
278
279     /* 将写文件所用到的块地址收集到 all_blocks，系统中块大小等于扇区大小，
280      * 后面都统一在 all_blocks 中获取写入扇区地址 */
281       if (add_blocks == 0) {
282       /* 在同一扇区内写入数据，不涉及到分配新扇区 */
283           if (file_will_use_blocks <= 12 ) {  // 文件数据量将在 12 块之内
284               block_idx = file_has_used_blocks - 1;
            // 指向最后一个已有数据的扇区
285               all_blocks[block_idx] = file->fd_inode->i_sectors[block_idx];
```

```
286          } else {
287      /* 未写入新数据之前已经占用了间接块，需要将间接块地址读进来 */
288              ASSERT(file->fd_inode->i_sectors[12] != 0);
289              indirect_block_table = file->fd_inode->i_sectors[12];
290              ide_read(cur_part->my_disk, indirect_block_table, \
         all_blocks + 12, 1);
291          }
292      } else {
293      /* 若有增量，便涉及到分配新扇区及是否分配一级间接块表，
         下面要分三种情况处理 */
294      /* 第一种情况:12 个直接块够用*/
295          if (file_will_use_blocks <= 12 ) {
296          /* 先将有剩余空间的可继续用的扇区地址写入 all_blocks */
297              block_idx = file_has_used_blocks - 1;
298              ASSERT(file->fd_inode->i_sectors[block_idx] != 0);
299              all_blocks[block_idx] = file->fd_inode->i_sectors[block_idx];
300
301          /* 再将未来要用的扇区分配好后写入 all_blocks */
302              block_idx = file_has_used_blocks; // 指向第一个要分配的新扇区
303              while (block_idx < file_will_use_blocks) {
304                  block_lba = block_bitmap_alloc(cur_part);
305                  if (block_lba == -1) {
306                      printk("file_write: block_bitmap_alloc
                         for situation 1 failed\n");
307                      return -1;
308                  }
309
310          /* 写文件时，不应该存在块未使用，但已经分配扇区的情况，
          当文件删除时，就会把块地址清 0 */
311                  ASSERT(file->fd_inode->i_sectors[block_idx] == 0);
          // 确保尚未分配扇区地址
312                  file->fd_inode->i_sectors[block_idx] = \
                 all_blocks[block_idx] = block_lba;
313
314                  /* 每分配一个块就将位图同步到硬盘 */
315                  block_bitmap_idx = block_lba - cur_part->sb->data_start_lba;
316                  bitmap_sync(cur_part, block_bitmap_idx, BLOCK_BITMAP);
317
318                  block_idx++;   // 下一个分配的新扇区
319            }
320          } else if (file_has_used_blocks <= 12 && file_will_use_blocks > 12) {
321          /* 第二种情况: 旧数据在 12 个直接块内，新数据将使用间接块*/
322
323          /* 先将有剩余空间的可继续用的扇区地址收集到 all_blocks */
324              block_idx = file_has_used_blocks - 1;
              // 指向旧数据所在的最后一个扇区
325              all_blocks[block_idx] = file->fd_inode->i_sectors[block_idx];
326
327              /* 创建一级间接块表 */
328              block_lba = block_bitmap_alloc(cur_part);
329              if (block_lba == -1) {
330                  printk("file_write: block_bitmap_alloc
                 for situation 2 failed\n");
331                  return -1;
332              }
333
334              ASSERT(file->fd_inode->i_sectors[12] == 0);
             // 确保一级间接块表未分配
335              /* 分配一级间接块索引表 */
336              indirect_block_table = file->fd_inode->i_sectors[12] = block_lba;
337
338              block_idx = file_has_used_blocks;
             // 第一个未使用的块，即本文件最后一个已经使用的直接块的下一块

339              while (block_idx < file_will_use_blocks) {
340                  block_lba = block_bitmap_alloc(cur_part);
341                  if (block_lba == -1) {
342                      printk("file_write: block_bitmap_alloc
                     for situation 2 failed\n");
343                      return -1;
344                  }
345
```

```
346             if (block_idx < 12) {
                 // 新创建的 0～11 块直接存入 all_blocks 数组
347                 ASSERT(file->fd_inode->i_sectors[block_idx] == 0);
                      // 确保尚未分配扇区地址

348                 file->fd_inode->i_sectors[block_idx] = \
                    all_blocks[block_idx] = block_lba;
349             } else {
                // 间接块只写入到 all_block 数组中，待全部分配完成后一次性同步到硬盘
350                 all_blocks[block_idx] = block_lba;
351             }
352
353             /* 每分配一个块就将位图同步到硬盘 */
354             block_bitmap_idx = block_lba - cur_part->sb->data_start_lba;
355             bitmap_sync(cur_part, block_bitmap_idx, BLOCK_BITMAP);
356
357             block_idx++;   // 下一个新扇区
358         }
359         ide_write(cur_part->my_disk, indirect_block_table, \
                      all_blocks + 12, 1);  // 同步一级间接块表到硬盘
360     } else if (file_has_used_blocks > 12) {
361         /* 第三种情况:新数据占据间接块*/
362         ASSERT(file->fd_inode->i_sectors[12] != 0);
            // 已经具备了一级间接块表
363         indirect_block_table = file->fd_inode->i_sectors[12];
                         // 获取一级间接表地址
364
365     /* 已使用的间接块也将被读入 all_blocks，无需单独收录 */
366         ide_read(cur_part->my_disk, indirect_block_table, \
                     all_blocks + 12, 1);   // 获取所有间接块地址
367
368         block_idx = file_has_used_blocks;
            // 第一个未使用的间接块，即已经使用的间接块的下一块

369         while (block_idx < file_will_use_blocks) {
370            block_lba = block_bitmap_alloc(cur_part);
371            if (block_lba == -1) {
372                printk("file_write: block_bitmap_alloc
               for situation 3 failed\n");
373                return -1;
374            }
375            all_blocks[block_idx++] = block_lba;
376
377            /* 每分配一个块就将位图同步到硬盘 */
378            block_bitmap_idx = block_lba - cur_part->sb->data_start_lba;
379            bitmap_sync(cur_part, block_bitmap_idx, BLOCK_BITMAP);
380         }
381         ide_write(cur_part->my_disk, indirect_block_table, \
            all_blocks + 12, 1);   // 同步一级间接块表到硬盘
382     }
383   }
384
385   /* 用到的块地址已经收集到 all_blocks 中，下面开始写数据 */
386   bool first_write_block = true;     // 含有剩余空间的块标识
387   file->fd_pos = file->fd_inode->i_size - 1;
      // 置 fd_pos 为文件大小-1，下面在写数据时随时更新
388   while (bytes_written < count) {     // 直到写完所有数据
389      memset(io_buf, 0, BLOCK_SIZE);
390      sec_idx = file->fd_inode->i_size / BLOCK_SIZE;
391      sec_lba = all_blocks[sec_idx];
392      sec_off_bytes = file->fd_inode->i_size % BLOCK_SIZE;
393      sec_left_bytes = BLOCK_SIZE - sec_off_bytes;
394
395      /* 判断此次写入硬盘的数据大小 */
396      chunk_size = size_left < sec_left_bytes ? size_left : sec_left_bytes;
397      if (first_write_block) {
398         ide_read(cur_part->my_disk, sec_lba, io_buf, 1);
399         first_write_block = false;
400      }
401      memcpy(io_buf + sec_off_bytes, src, chunk_size);
402      ide_write(cur_part->my_disk, sec_lba, io_buf, 1);
403      printk("file write at lba 0x%x\n", sec_lba);   // 调试，完成后去掉
```

```
404
405         src += chunk_size;   // 将指针推移到下个新数据
406         file->fd_inode->i_size += chunk_size;  // 更新文件大小
407         file->fd_pos += chunk_size;
408         bytes_written += chunk_size;
409         size_left -= chunk_size;
410      }
411      inode_sync(cur_part, file->fd_inode, io_buf);
412      sys_free(all_blocks);
413      sys_free(io_buf);
414      return bytes_written;
415 }
```

确实好长，小两百行。整个代码就这一个函数 file_write，它接受 3 个参数，文件 file、数据缓冲区 buf、字节数 count，功能是把 buf 中的 count 个字节写入 file，成功则返回写入的字节数，失败则返回−1。

前面说过，为实现省事，这里的块大小就是扇区大小。咱们的文件最大尺寸是 140 个块，为此在函数开头便判断新加的数据是否会使文件超过最大尺寸，如果超过了 140 个块的大小，即 BLOCK_SIZE * 140，程序打印提示后，返回−1，其中宏 BLOCK_SIZE 等于 SECTOR_SIZE，即值为 512。

因为后面我们的磁盘操作都以 1 个扇区为单位，所以申请了 512 字节的缓冲区给 io_buf。

为写硬盘时方便获取块地址，我们打算把文件所有的块地址收集到 all_blocks 中统一获取，为此第 235 行申请了 BLOCK_SIZE + 48 大小的内容，即 128 个间接块+12 个直接块的大小。

在第 241～252 行声明了一些变量，其意义都有相关注释，后面可以结合实际去理解，不解释啦。

和大伙儿交待下，文件中的数据是个整体，因此是顺序、连续写入块中的，并且是从最低块 i_sector[0]向高块开始写，比如直接块 0 写满后再写入直接块 1，写完直接块后再写第 0 个间接块，直到第 127 个间接块，数据从前往后写，后面的块都是空白的，在文件中也不会出现空洞、跨块的情况，这和目录是不同的，目录中的目录项是单独的个体，它们可以分散在不同的块中。

文件第一次写入数据时要为其分配块地址，若未分配块地址的话，块地址则为 0，这是之前调用 inode_init 为其初始化时提前安排的。第 255～267 行便判断文件是否是第一次写数据，如果是就通过 block_bitmap_alloc 分配扇区，地址写入文件的 i_sectors[0]，然后将位图同步到硬盘。

下面判断文件是否要为这 count 个字节分配新块，也就是现有的扇区是否够用。变量 file_has_used_blocks 是文件目前使用的块数，也就是未写入 count 个字节前文件占用的块数，变量 file_will_use_blocks 是存储 count 字节后该文件将占用的块数。变量 add_blocks 是需要为 count 个字节数据分配的扇区数。

接下来第 281～383 行是把文件现在使用及未来使用的块地址收集到 all_blocks 中（不包括那些不参与写入操作的块地址），与数据写入无关。收集工作完成之后，all_block 中包括原块地址及新数据占用的块地址，在这之后我们才进行数据写入工作。

第 281 行判断，如果 add_blocks 为 0，这说明 count 值小于等于原有扇区中的剩余空间，剩余空间便可容纳 count 个字节数据，无需再申请新的块。接着第 283～286 行判断原有块地址是否为直接块，第 287～291 行判断原有块地址是否是间接块，无论是直接块，还是间接块，文件的现有块地址都将收录到 all_blocks 中。

第 292 行是处理需要分配新数据块的情况，下面分三种情况讨论。

（1）若已经使用的扇区数在 12 块之内，新增了若干块后，文件大小还在 12 块之内，直接分配所需的块并把块地址写入 i_sectors 数组中即可。

（2）若已经使用的块数在 12 块之内，新增了若干块后，文件大小超过了 12 块，这种情况下所申请的块除了要写入 i_sector 数组，还要创建一级间接块表并写入块地址。

（3）若已经使用的扇区数超过了 12 块，这种情况下要在一级间接块表中创建间接块项，间接块项就是在一级间接块表中记录间接块地址的条目，姑且这么叫吧。

以上三种情况所申请的块地址都要收录到 all_blocks 中。

第 295～320 行是处理第一种情况：新分配的块也属于直接块的情况。这分为两步，第 1 步是先在第 297～299 行，将原有扇区中包含剩余空间的（可继续用的）块地址收录到 all_blocks。第 2 步是在第 302～319 行，再将未来要用的扇区分配好并写入 all_blocks。

第 320～360 行是处理第二种情况：旧数据在 12 个直接块内，新数据将使用间接块。这分为三步，先在第 324～325 行将原有扇区中包含剩余空间的块地址收录到 all_blocks。然后在第 328～336 行分配一个块作为一级间接块索引表。最后在第 338～358 行将未来要用的块分配好并收录到 all_blocks 中。接着在第 359 行将所有的间接块地址同步到硬盘上的一级间接块索引表中。

代码第 360～383 行处理第三种情况：新老数据都在间接块中。与前两种情况不同的是已使用的、包含剩余空间的间接块都在一级间接块索引表中，因此只要将此表读到 all_blocks 中便获取所有间接块地址，无需单独收录。第 368～380 行开始分配间接块并收录到 all_blocks 中。最后在第 381 行将所有间接块地址，包括新分配的间接块地址一同写入硬盘上的一级间接块索引表中。

到目前为止，all_blocks 中包含可继续使用的、含有剩余空间的块地址，以及新数据要占用的新的块地址。下面是开始接着在含有剩余空间的块写入新数据。

第 386 行的变量 first_write_block 用于标识本次写入操作中第一个所写入的块，除第一次写入数据外，通常情况下该块中都有些数据，最新的数据要从接着该块的空闲空间接着写。因此在第一次写数据时，要将该块中的数据读出来，将新数据写入该块中空闲区域，之后再新老数据一同写入硬盘，这样就保护好了老数据，并且实现了数据追加的功能。

第 388 行开始循环写入数据，变量 bytes_written 记录已写入的数据量大小，它已被初始为 0，循环直到 bytes_written 等于 count 时结束，也就是写完为止。循环体中最先执行的是用 memset 清空 io_buf，io_buf 是要往文件数据块写入数据的缓冲区，必须要保证干净。

第 390 行的 sec_idx 是根据文件大小计算出的块索引，也就是扇区索引值，在下一行将它代入 all_blocks 获得扇区地址 sec_lba。sec_off_bytes 是数据在最后一个块中的偏移量，sec_left_bytes 是块中的可用字节空间。

第 396 行的 chunk_size 是本次写入扇区的数据量，size_left 表示剩下未写入硬盘的数据量，当 size_left 小于本次所写入块的剩余空间时，chunk_size 就等于 size_left，这属于剩余的数据不足一个块，往最后一个块中写剩余数据时的情况，如果 size_left 大于等于 sec_left_bytes，那就把剩下的空闲空间写满，即 chunk_size 等于 sec_left_bytes。

第 397 行判断，若马上要读写的块是本次操作中的第一个块，通常情况下该块中都已存在数据，因此在下一行先将该块中的数据读出来，然后使 first_write_block 置为 false。第 401 行将数据拷贝到 io_buf 中，拼好数据，在下一行将其写入硬盘。第 403 行的 printk 输出信息是为了帮助调试，下次就将它去掉。硬盘写入后，第 405～409 行更新相应的数据，如使 i_size 加上 chunk_size，bytes_written 加上 bytes_written，剩余数据量减去 bytes_written。

将 count 个字节写完之后，在第 411 行同步 inode。最后释放 all_blocks 和 io_buf，返回已写入的数据量 bytes_written，至此 file_write 结束。

虽然就这一个函数，内容还真是不少，大家辛苦了，休息一下，然后继续。

14.7.2　改进 sys_write 及 write 系统调用

本节给大伙儿带来一个好消息和一个坏消息，坏消息是我们要重写 sys_write 及周边代码了，好消息是 sys_wirte 非常简单，相关改动也不麻烦，大伙儿忍一忍，本节下面介绍，见代码 14-26。

代码 14-26　（project/c14/e/fs/fs.c）

```
…略
383 /* 将 buf 中连续 count 个字节写入文件描述符 fd，
    成功则返回写入的字节数，失败返回-1 */
384 int32_t sys_write(int32_t fd, const void* buf, uint32_t count) {
385     if (fd < 0) {
386         printk("sys_write: fd error\n");
387         return -1;
388     }
389     if (fd == stdout_no) {
390         char tmp_buf[1024] = {0};
391         memcpy(tmp_buf, buf, count);
392         console_put_str(tmp_buf);
```

```
393         return count;
394     }
395     uint32_t _fd = fd_local2global(fd);
396     struct file* wr_file = &file_table[_fd];
397     if (wr_file->fd_flag & O_WRONLY || wr_file->fd_flag & O_RDWR) {
398         uint32_t bytes_written  = file_write(wr_file, buf, count);
399         return bytes_written;
400     } else {
401         console_put_str("sys_write: not allowed to write file
            without flag O_RDWR or O_WRONLY\n");
402         return -1;
403     }
404 }
…略
```

sys_write 接受 3 个参数，文件描述符 fd、数据所在缓冲区 buf、写入的字节数 count。

函数中对标准输出做了特殊处理，第 389 行，若发现 fd 等于 stdout_no，也就是往屏幕上打印信息时，就把 buf 中的 count 字节复制到临时缓冲区 tmp_buf 中，然后调用 console_put_str(tmp_buf)输出，最后通过 return 返回 count。

在其他情况下，sys_write 都是往文件中写数据，下面第 395 行通过 fd_local2global 获取文件描述符 fd 对应于文件表中的下标_fd，然后获得待写入文件的文件结构指针 wr_file，在第 397 行判断其 flag，只有 flag 包含 O_WRONLY 或 O_RDWR 的文件才允许写入数据，提醒一下，单纯的 O_CREAT 只能创建文件，不能写文件，咱们这里是参照 Linux 的做法实现的。如果符合条件，则通过 file_write 完成数据写入，并返回写入的字节数，否则不允许写入数据，输出提示信息后返回−1。

好啦，sys_write 修改完成，下面是修改和此函数相关的功能。

之前咱们的系统调用 write 对应的 sys_write 还是非常简单的，那时咱们还不支持文件系统，因此 sys_write 就只是调用 console_put_str 完成字符串输出。现在 sys_write 已经改版了，原型不同，参数个数不同了，因此和 write 相应的都要一同改进，真是牵一发而动全身啊。

首先改进的是 lib/user/syscall.c，见代码 14-27。

代码 14-27 （project/c14/e/lib/user/syscall.c）

```
…略
56 /* 把 buf 中 count 个字符写入文件描述符 fd */
57 uint32_t write(int32_t fd, const void* buf, uint32_t count) {
58     return _syscall3(SYS_WRITE, fd, buf, count);
59 }
…略
```

write 改动很小，之前是调用宏_syscall1，现在改为了宏_syscall3。另外记得把 write 的声明在头文件 lib/user/syscall.h 中更新。

下个要改进的是 lib/stdio.c，见代码 14-28。

代码 14-28 （project/c14/e/lib/stdio.c）

```
…略
84 /* 格式化输出字符串 format */
85 uint32_t printf(const char* format, ...) {
86     va_list args;
87     va_start(args, format);           // 使 args 指向 format
88     char buf[1024] = {0};             // 用于存储拼接后的字符串
89     vsprintf(buf, format, args);
90     va_end(args);
91     return write(1, buf, strlen(buf));
92 }
…略
```

这里把 printf 中调用的 write 也替换了，改为 write(1, buf, strlen(buf))，之前是 write(buf)。

接着把原定义在 usrprog/syscall-init.c 中的 sys_write 去掉就好了。

好啦，周边代码就改好了，本节到这就结束了，下节咱们进行功能测试。

14.7.3　把数据写入文件

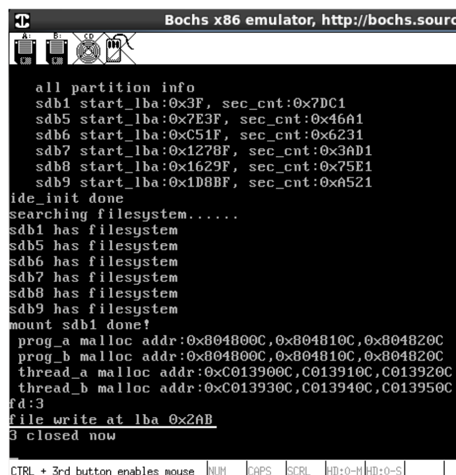
本节咱们往文件 file1 中写入数据，在 main.c 中添加的测试代码见代码 14-29。

代码 14-29　（project/c14/e/kernel/main.c）

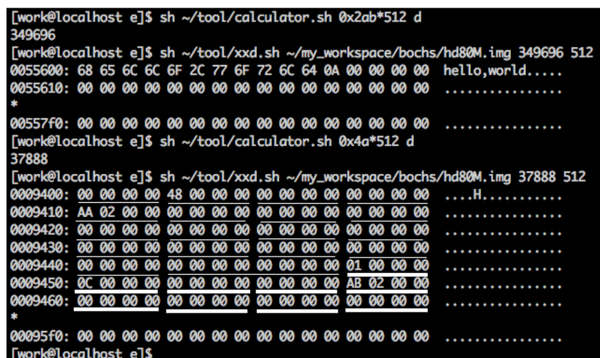
```
…略
18 int main(void) {
19      put_str("I am kernel\n");
20      init_all();
21      process_execute(u_prog_a, "u_prog_a");
22      process_execute(u_prog_b, "u_prog_b");
23      thread_start("k_thread_a", 31, k_thread_a, "I am thread_a");
24      thread_start("k_thread_b", 31, k_thread_b, "I am thread_b");
25
26      uint32_t fd = sys_open("/file1", O_RDWR);
27      printf("fd:%d\n", fd);
28      sys_write(fd, "hello,world\n", 12);
29      sys_close(fd);
30      printf("%d closed now\n", fd);
31      while(1);
32      return 0;
33 }
…略
```

代码第 26 行以 O_RDWR 方式打开文件 file1，然后在第 28 行写入“hello,world\n”，下面我们看下程序运行的结果，如图 14-24 所示。

图中标有下画线的部分是我们在程序中添加的调试代码，输出写入的扇区地址是 0x2ab，下面我们查看一下该地址处的数据，还是用 xxd 工具。另外，我们顺便查看下文件 file1 的 inode 的信息，在图 14-18 中有 inode_table 的地址，该地址是 0x4a，这两类信息如图 14-25 所示。



▲图 14-24　文件写入演示



▲图 14-25　inode 及文件数据验证

图中上半部分是查看文件写入的数据。将扇区地址 0x2ab 换算为字节是 349696，用 xxd 脚本查看 hd80M.img 的该地址处的 1 扇区，xxd 在右边显示了字符“hello,world”，但'\n'是不可见字符，因此只是以'.'代替，大伙儿可以对照下左边同一行的 0A，它就是'\n'的 ASCII 码。

图中下半部分是查看 inode_table 中的前两个 inode，我们目的是查看 inode 的 i_size 和 i_sectors 中的地址。这里只显示了 512 字节，因此第二个 inode 信息不完整，但已经足够了。图中用细线标出的是第 0 个 inode，即根目录的 inode，地址范围是 0x9400～0x944b，每个 inode 的都是这么大。最后较长的下画线标出了 8 个字节，它是 inode 中的 inode_tag，类型为 struct list_elem，前 4 字节是指针 prev，后 4 字节是指针 next。其余的数据都是按每 4 字节为一个单位，第一行的第 1 个 4 字节表示 inode 编号 i_no，其值为 0，根目录的 inode 编号就是 0。第 2 个 4 字节表示 i_size，此 inode 表示目录，因此 i_size 是目录中目录项大小的总和，大家可以看

下目录项的结构，其大小是 24 字节，根目录中有三个目录项，分别是'.'、'..'和'file1'，因此根目录 inode 的 i_size 是 72，即 0x48。第 3 个 4 字节和第 4 个 4 字节分别表示 i_open_cnts 和 write_deny，它们在写入硬盘时就被清 0 了。第二行的第 1 个 4 字节是根目录的 i_sectors[0]的值，值为 0x2aa，咱们 file1 的数据块紧邻其后，是 0x2ab。此 inode 其余的是 i_sector[1～12]的值，都为 0。图中画粗下画线的是 file1 的 inode，第 1 个粗线表示 i_no，即 inode 编号为 1。第 2 个粗线表示 i_size，其值为 0xc，即 12，咱们确实写入了 12 个字节的数据。第 5 个粗线表示 i_sector[0]，其地址确实是 0x2ab，符合预期。

好啦，下面再执行一次程序，看看数据是否更新正确，由于还是写入 12 个字符，未超过 1 扇区，故 bochs 的运行结果和图 14-24 一样，不重复贴图了，硬盘上的数据如图 14-26 所示。

经过再次执行一次写入，file1 的内容变成了两个“hello,world\n”，另外 inode 中的 i_size 变为了 0x18，即 24 字节，符合预期。

好啦，收工。

```
[work@localhost e]$ sh ~/tool/xxd.sh ~/my_workspace/bochs/hd80M.img 349696 512
0055600: 68 65 6C 6C 6F 2C 77 6F 72 6C 64 0A 68 65 6C 6C  hello,world.hell
0055610: 6F 2C 77 6F 72 6C 64 0A 00 00 00 00 00 00 00 00  o,world.........
0055620: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00557f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[work@localhost e]$ sh ~/tool/xxd.sh ~/my_workspace/bochs/hd80M.img 37888 512
0009400: 00 00 00 00 48 00 00 00 00 00 00 00 00 00 00 00  ....H...........
0009410: AA 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0009420: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0009430: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0009440: 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00  ................
0009450: 18 00 00 00 00 00 00 00 00 00 00 00 AB 02 00 00  ................
0009460: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00095f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[work@localhost e]$
```

▲图 14-26 数据变化

14.8 读取文件

上节完成了写入文件，这下巧妇也有米可炊了，现在咱们得想办法把数据读出来，完成读取文件的功能。

14.8.1 实现 file_read

文件读取的核心函数还是定义在 file.c 中，这次咱们添加函数 file_read，见代码 14-30。

代码 14-30 （project/c14/f/fs/file.c）

```
…略
416 /* 从文件 file 中读取 count 个字节写入 buf,
返回读出的字节数，若到文件尾则返回-1 */
417 int32_t file_read(struct file* file, void* buf, uint32_t count) {
418     uint8_t* buf_dst = (uint8_t*)buf;
419     uint32_t size = count, size_left = size;
420
421     /* 若要读取的字节数超过了文件可读的剩余量,
        就用剩余量作为待读取的字节数 */
422     if ((file->fd_pos + count) > file->fd_inode->i_size) {
423         size = file->fd_inode->i_size - file->fd_pos;
424         size_left = size;
425         if (size == 0) {     // 若到文件尾，则返回-1
426             return -1;
427         }
428     }
429
430     uint8_t* io_buf = sys_malloc(BLOCK_SIZE);
431     if (io_buf == NULL) {
432         printk("file_read: sys_malloc for io_buf failed\n");
433     }
434     uint32_t* all_blocks = (uint32_t*)sys_malloc(BLOCK_SIZE + 48);
       // 用来记录文件所有的块地址
435     if (all_blocks == NULL) {
436         printk("file_read: sys_malloc for all_blocks failed\n");
437         return -1;
438     }
439
440     uint32_t block_read_start_idx = file->fd_pos / BLOCK_SIZE;
                    // 数据所在块的起始地址

441     uint32_t block_read_end_idx = (file->fd_pos + size) / BLOCK_SIZE;
        // 数据所在块的终止地址

442     uint32_t read_blocks = block_read_start_idx - block_read_end_idx;
```

```
               // 如增量为 0，表示数据在同一扇区
443      ASSERT(block_read_start_idx < 139 && block_read_end_idx < 139);
444
445      int32_t indirect_block_table;      // 用来获取一级间接表地址
446      uint32_t block_idx;                // 获取待读的块地址
447
448  /* 以下开始构建 all_blocks 块地址数组，
     专门存储用到的块地址（本程序中块大小同扇区大小）*/
449      if (read_blocks == 0) {  // 在同一扇区内读数据，不涉及到跨扇区读取
450          ASSERT(block_read_end_idx == block_read_start_idx);
451          if (block_read_end_idx < 12 ) {   // 待读的数据在 12 个直接块之内
452              block_idx = block_read_end_idx;
453              all_blocks[block_idx] = file->fd_inode->i_sectors[block_idx];
454          } else {   // 若用到了一级间接块表，需要将表中间接块读进来
455              indirect_block_table = file->fd_inode->i_sectors[12];
456              ide_read(cur_part->my_disk, \
                 indirect_block_table, all_blocks + 12, 1);
457          }
458      } else {      // 若要读多个块
459      /* 第一种情况：起始块和终止块属于直接块*/
460          if (block_read_end_idx < 12 ) {    // 数据结束所在的块属于直接块
461              block_idx = block_read_start_idx;
462              while (block_idx <= block_read_end_idx) {
463                 all_blocks[block_idx] = file->fd_inode->i_sectors[block_idx];
464                 block_idx++;
465              }
466          } else if (block_read_start_idx < 12 && block_read_end_idx >= 12) {
467      /* 第二种情况：待读入的数据跨越直接块和间接块两类*/
468          /* 先将直接块地址写入 all_blocks */
469              block_idx = block_read_start_idx;
470              while (block_idx < 12) {
471                  all_blocks[block_idx] = file->fd_inode->i_sectors[block_idx];
472                  block_idx++;
473              }
474              ASSERT(file->fd_inode->i_sectors[12] != 0);
             // 确保已经分配了一级间接块表
475
476              /* 再将间接块地址写入 all_blocks */
477              indirect_block_table = file->fd_inode->i_sectors[12];
478              ide_read(cur_part->my_disk, \
              indirect_block_table, all_blocks + 12, 1);
                        // 将一级间接块表读进来写入到第 13 个块的位置之后
479          } else {
480      /* 第三种情况：数据在间接块中*/
481              ASSERT(file->fd_inode->i_sectors[12] != 0);
         // 确保已经分配了一级间接块表
482              indirect_block_table = file->fd_inode->i_sectors[12];
          // 获取一级间接表地址
483              ide_read(cur_part->my_disk, \
              indirect_block_table, all_blocks + 12, 1);
                      // 将一级间接块表读进来写入到第 13 个块的位置之后
484          }
485      }
486
487      /* 用到的块地址已经收集到 all_blocks 中，下面开始读数据 */
488      uint32_t sec_idx, sec_lba, sec_off_bytes, sec_left_bytes, chunk_size;
489      uint32_t bytes_read = 0;
490      while (bytes_read < size) {       // 直到读完为止
491         sec_idx = file->fd_pos / BLOCK_SIZE;
492         sec_lba = all_blocks[sec_idx];
493         sec_off_bytes = file->fd_pos % BLOCK_SIZE;
494         sec_left_bytes = BLOCK_SIZE - sec_off_bytes;
495         chunk_size = size_left < sec_left_bytes ? size_left : sec_left_bytes;
                                                    // 待读入的数据大小
496
497         memset(io_buf, 0, BLOCK_SIZE);        // 不清空也可以
498         ide_read(cur_part->my_disk, sec_lba, io_buf, 1);
499         memcpy(buf_dst, io_buf + sec_off_bytes, chunk_size);
500
501         buf_dst += chunk_size;
502         file->fd_pos += chunk_size;
503         bytes_read += chunk_size;
```

```
504         size_left -= chunk_size;
505      }
506      sys_free(all_blocks);
507      sys_free(io_buf);
508      return bytes_read;
509 }
```

file_read 同 file_write 一样，都是比较长，实现原理也类似，下面大体说下。

file_read 接受 3 个参数，读取的文件 file、数据写入的缓冲区 buf、读取的字节数 count，功能是从文件 file 中读取 count 个字节写入 buf，返回读出的字节数，若到文件尾，则返回−1。

函数开头将 buf_dst 用 buf 赋值，后面我们将读到的数据存入此地址，不改变 buf。代码第 422～428 行判断文件是否已读到文件尾，如果是，就返回−1。

老套路，第 430 行的 io_buf 还是咱们硬盘操作的缓冲区，后面会把从硬盘中读出的数据存入到 io_buf。第 434 行的 all_blocks 依然用来记录文件所有的块地址，我们后面的读硬盘操作将在此结构中获取地址。

变量 block_read_start_idx 表示当前指针 fd_pod 所指向的块索引，也就是数据读取的起始块索引，block_read_end_idx 表示相对于当前位置 fd_pos 偏移 count 个字节所在的块索引，即数据读取的结束块索引。read_blocks 表示要读取的块数。indirect_block_table 是一级间接块索引表的地址，后面将为它赋值，block_idx 用于块索引。

下面开始把读操作中用到的块地址收录到 all_blocks 中。

当 read_blocks 为 0 时，这说明要读取的 count 个字节在一个块中，因此只需要读取 1 个块，下面分两种情况处理。第 451 行判断若结束块属于直接块，就在 i_sectors 中获取块地址。否则就要获得间接块地址，因此第 455 行获取一级间接块索引表地址，第 456 行读取该表，获取所有的间接块地址。

若 read_blocks 不为 0，这说明 count 个字节跨块，需要读取多个块，下面分三种情况处理。

（1）若起始块和终止块都在 12 块之内，直接读入 i_sectors 数组中即可。

（2）若起始块在 12 块之内，结束块超过了 12 块，除了要读入 i_sector 数组，还要从一级间接块索引表中读取间接块地址。

（3）若起始块超过了 12 块，这种情况下要在一级间接块索引表中读取间接块。

第 460～465 行处理第（1）种情况，循环将所需要的间接块地址录入 all_blocks。

第 466～478 行处理第（2）种情况，先在第 469～473 行通过 while 循环收集直接块地址，然后在第 477～478 行从硬盘上获取所有的间接块地址，录入到 all_blocks 中。

第 479～483 行是直接从硬盘上获取间接块地址到 all_blocks。

至此，读硬盘操作中所涉及的块地址已经录入到 all_blocks 中。

第 490～505 行是读取硬盘的过程，原理同写入数据类似，都是要选出合适的操作数大小，即 chunk_size，每次都由第 498 行的 ide_read 函数读取 1 个扇区，然后在第 499 行往 dst_buf 中拷贝 chunk_size 个字节。顺便提一下，第 497 行的 memset 清 0 操作不是必须的，理论上读入的数据会把 io_buf 中旧数据覆盖，而且最终返回给用户的数据是在 dst_buf 中，这由第 499 行的 memcpy 控制，数据读取的准确性是由 chunk_size 把控的，因此即使 io_buf 中有垃圾数据，只要 chunk_size 正确就不会多读入错误的数据。不过惭愧的是自我安慰了这么多还是把 memset 放在这，毕竟小心驶得万年船，没有人是故意犯错的，错误总是由想不到的原因引起的嘛。

最后分别释放 all_blocks 和 io_buf，通过 return 返回已读的字节数 bytes_read。

本节我们也结束了，下节再继续。

14.8.2 实现 sys_read 与功能验证

sys_read 是 file_read 的封装，主要功能已经由 file_read 完成了，因此给大伙儿带来了好消息，sys_read 代码非常少，不信您看，见代码 14-31。

代码 14-31　（project/c14/f/fs/fs.c）

```
…略
406 /* 从文件描述符 fd 指向的文件中读取 count 个字节到 buf,
   若成功则返回读出的字节数，到文件尾则返回-1 */
407 int32_t sys_read(int32_t fd, void* buf, uint32_t count) {
408     if (fd < 0) {
409         printk("sys_read: fd error\n");
410         return -1;
411     }
412     ASSERT(buf != NULL);
413     uint32_t _fd = fd_local2global(fd);
414     return file_read(&file_table[_fd], buf, count);
415 }
…略
```

sys_read 是 read 系统调用的内核实现，原型也是按照 read 来实现的，目前就改成这样，以后还会再加标准输入。没啥好说的啦，代码简单明了，下面在 main.c 中增加 sys_read 调用，见代码 14-32。

代码 14-32　（project/c14/f/kernel/main.c）

```
…略
19  int main(void) {
…略
27     uint32_t fd = sys_open("/file1", O_RDWR);
28       printf("open /file1,fd:%d\n", fd);
29       char buf[64] = {0};
30       int read_bytes = sys_read(fd, buf, 18);
31       printf("1_ read %d bytes:\n%s\n", read_bytes, buf);
32
33       memset(buf, 0, 64);
34       read_bytes = sys_read(fd, buf, 6);
35       printf("2_ read %d bytes:\n%s", read_bytes, buf);
36
37       memset(buf, 0, 64);
38       read_bytes = sys_read(fd, buf, 6);
39       printf("3_ read %d bytes:\n%s", read_bytes, buf);
40
41       printf("________  close file1 and reopen  ________\n");
42       sys_close(fd);
43       fd = sys_open("/file1", O_RDWR);
44       memset(buf, 0, 64);
45       read_bytes = sys_read(fd, buf, 24);
46       printf("4_ read %d bytes:\n%s", read_bytes, buf);
47
48       sys_close(fd);
49       while(1);
50       return 0;
51 }
…略
```

我们之前已经在文件 file1 中写入了 24 个字节，即两组字符串“hello,world\n”，一共 24 个字符，下面将其“曲折地”读出来，考验我们的时候到了。

测试代码分为 4 块，第 27～31 行是打开 file1，读取 18 个字节到 buf 中，然后输出读到的数据。按理来说，这 18 个字符应该是“hello,world\nhello,”，其中的'\n'会被处理为换行。

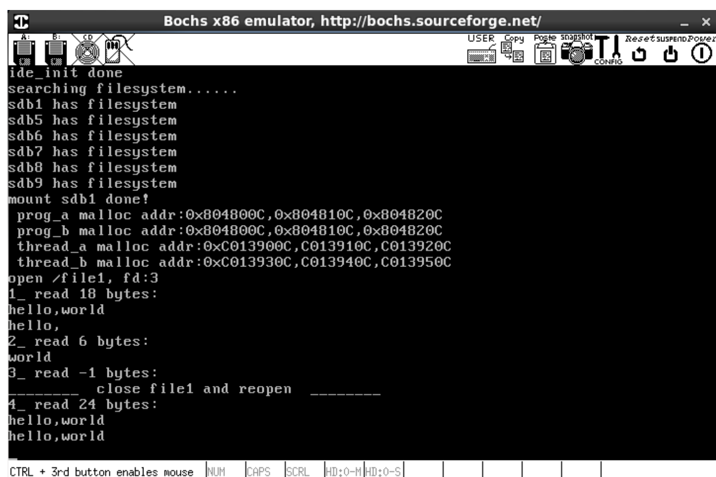
第 34～35 行是继续读取剩下的 6 字节并输出，按理来说会输出“world\n”，至此 24 个字符全部读完，已读到文件末尾。

第 38～39 行是再读取 6 字节。按理说，由于目前文件已读到文件末尾，再无数据可读取，sys_read 会返回−1。

第 41～46 行关闭文件 file1，再重新打开，目的是测试是否能从头读取数据。不过提醒一句，file1 不一定要先关闭，完全可以直接打开，但要用另一变量记录新的文件描述符，否则之前的 fd 就丢掉了，无法关闭。

这次我们读取 24 个字节。运行结果如图 14-27 所示。

结果如图 14-27 所示，还是符合预期的，本节任务完成。



▲图 14-27　文件读取测试

14.9　实现文件读写指针定位功能

本节的名字看上去还挺炫的，又是文件指针，又是定位的，哈哈，其实就是系统调用 lseek 的内核实现，您懂的，本质上就是设置文件读写时的起始偏移量。

为什么要实现这个功能呢？您看，上次咱们读取 file1 时，由于已经读到了文件末尾，sys_read 返回了−1，为了从头读取文件，咱们还特意把文件关闭重新打开一次，这太麻烦了，我们必须提供不关闭文件就能自由设置读写位置的方法。

为了统一，咱们系统中的系统调用对应的内核实现，都是在系统调用名前加上 sys_前缀，因此我们本节要实现的就是 sys_lseek。我们先看下 lseek 函数的用法吧，咱们要传承标准接口。

lseek 原型是“off_t lseek(int fd, off_t offset, int whence)”，fd 是文件描述符，offset 是偏移量，whence 是 offset 的“参照物”，函数功能是设置文件读写指针 fd_pos 为参照物+偏移量的值，也就是说，文件指针具体的位置不仅取决于 offset，还取决于 whence。其中 off_t 是 typedef 自定义的类型，相当于 signed int，有符号整型，因此 offset 是可正可负的值。

whence 取值有三种取值。

SEEK_SET，offset 的参照物是文件开始处，也就是将读写位置指针设置为距文件开头偏移 offset 个字节处。

SEEK_CUR，offset 的参照物是当前读写位置，也就是将读写位置指针设置为当前位置+offset。

SEEK_END，offset 的参照物是文件尺寸大小，即文件最后一个字节的下一个字节，也就是将读写位置指针设置为文件尺寸+offset。

这里说一下 SEEK_END，也许您觉得 offset 的参照物应该是文件的最后一个字节，很遗憾这是“错觉”，因为这不是文件末尾，文件末尾是指文件最后一个字节的下一个字节处，即超出文件大小的第 1 个字节，这就是读完文件时很多函数都会返回 EOF（-1）的原因，不属于文件范围了嘛。文件的读写位置指针是 fd_pos，fd_pos 始终指向下一个可读写的位置，它是以 0 为起始的偏移量，因此文件末尾是指文件大小。

好啦，咱们要动手实现这个功能了。咱们先在 fs.h 中增加 whence 的结构，见代码 14-33。

代码 14-33　（project/c14/g/fs/fs.h）

```
…略
27 /* 文件读写位置偏移量 */
28 enum whence {
29     SEEK_SET = 1,
30     SEEK_CUR,
31     SEEK_END
32 };
…略
```

代码很简单，同以往不同的是枚举类型 whence 中的第 1 个成员是从 1 开始的，即 SEEK_SET=1，另外两个递增，分别是 2 和 3。

下面看 sys_lseek 的实现，它定义在 fs.c 中，见代码 14-34。

代码 14-34　（project/c14/g/fs/fs.c）

```
…略
417 /* 重置用于文件读写操作的偏移指针。
   成功时返回新的偏移量，出错时返回-1 */
418 int32_t sys_lseek(int32_t fd, int32_t offset, uint8_t whence) {
419    if (fd < 0) {
420        printk("sys_lseek: fd error\n");
421        return -1;
422    }
423    ASSERT(whence > 0 && whence < 4);
424    uint32_t _fd = fd_local2global(fd);
425    struct file* pf = &file_table[_fd];
426    int32_t new_pos = 0;     //新的偏移量必须位于文件大小之内
427    int32_t file_size = (int32_t)pf->fd_inode->i_size;
428    switch (whence) {
429 /* SEEK_SET 新的读写位置是相对于文件开头再增加 offset 个位移量 */
430          case SEEK_SET:
431             new_pos = offset;
432             break;
433
434  /* SEEK_CUR 新的读写位置是相对于当前的位置增加 offset 个位移量 */
435          case SEEK_CUR:    // offse 可正可负
436             new_pos = (int32_t)pf->fd_pos + offset;
437             break;
438
439  /* SEEK_END 新的读写位置是相对于文件尺寸再增加 offset 个位移量 */
440          case SEEK_END:      // 此情况下，offset 应该为负值
441             new_pos = file_size + offset;
442    }
443    if (new_pos < 0 || new_pos > (file_size - 1)) {
444          return -1;
445    }
446    pf->fd_pos = new_pos;
447    return pf->fd_pos;
448 }
…略
```

sys_lseek 接受 3 个参数，文件描述符 fd、偏移量 offset、参数位置 whence，功能是重置用于文件读写操作的偏移指针，成功时返回新的偏移量，出错时返回−1。文件的读写位置是由文件结构中的 fd_pos 决定的，因此 sys_lseek 的原理是将 whence+offset 转换为 fd_pos。

函数开头先将文件描述符 fd 通过 fd_local2global 转换为文件表中的下标，然后在第 425 行用 pf 指向对应的文件结构，后面都用指针 pf 来操作啦。接着声明变量 new_pos 作为新的 fd_pos 值，后面会根据不同的情况计算该值。file_size 是文件的大小，下面通过 switch 结构判断参照物 whence 的情况。

位置指针以 0 为起始偏移量，无论 whence 是 SEEK_SET、SEEK_CUR，还是 SEEK_END，新位置指针 new_pos 肯定要在文件大小范围内，不能小于 0，也不能超过文件尺寸−1，即不能大于等于文件尺寸。第 428～442 行分别计算了三种情况下的 new_pos，代码意义如之前的说明，很简单就不说了。

第 443 行判断新的位置 new_pos 是否在文件大小范围之外，如果是就返回−1，否则就在第 446 行用 new_pos 修改文件的 fd_pos，然后返回新的位置。

sys_lseek 介绍完了，下面咱们修改下 main.c，用 sys_lseek 替换之前那个麻烦的“重新打开”，见代码 14-35。

代码 14-35　（project/c14/g/kernel/main.c）

```
…略
19 int main(void) {
…略
27     uint32_t fd = sys_open("/file1", O_RDWR);
28      printf("open /file1, fd:%d\n", fd);
29      char buf[64] = {0};
```

```
30      int read_bytes = sys_read(fd, buf, 18);
31      printf("1_ read %d bytes:\n%s\n", read_bytes, buf);
32
33      memset(buf, 0, 64);
34      read_bytes = sys_read(fd, buf, 6);
35      printf("2_ read %d bytes:\n%s", read_bytes, buf);
36
37      memset(buf, 0, 64);
38      read_bytes = sys_read(fd, buf, 6);
39      printf("3_ read %d bytes:\n%s", read_bytes, buf);
40
41      printf("_______  SEEK_SET 0  _______\n");
42      sys_lseek(fd, 0, SEEK_SET);
43      memset(buf, 0, 64);
44      read_bytes = sys_read(fd, buf, 24);
45      printf("4_ read %d bytes:\n%s", read_bytes, buf);
46
47      sys_close(fd);
48      while(1);
49      return 0;
50 }
…略
```

第 40 行之前的代码没变过，这里用第 42 行的“sys_lseek(fd, 0, SEEK_SET)”代替了之前的先 close 再重新 open，功能是把读写位置指针置为文件开头偏移为 0 的地址，也就是置为文件开头。好啦，看下运行结果，如图 14-28 所示。

▲图 14-28　lseek 验证

图中最后输出了两行“hello,world\n”，这说明 sys_lseek 功能符合预期，另外的 SEEK_CUR 和 SEEK_END 就不测试了，我私下测试还行，大伙儿有兴趣自己测试吧。

好啦，今天就到这了，很晚了，累，大伙儿晚安。

14.10 实现文件删除功能

在 Linux 下用于文件删除的函数是 unlink，咱们本节就要实现它。删除文件是创建文件的逆过程，会涉及到 inode、inode 位图、目录 inode 中的 i_size、目录项、数据块及数据块位图的回收操作，因此还是先构建“底层建筑”。

14.10.1 回收 inode

inode 是文件系统的灵魂，删除文件最重要的就是回收文件对应的 inode。与 inode 相关的资源有：

（1）inode 位图

（2）inode_table

（3）inode 中 i_sectors[0～11]中的直接块和一级间接索引块表 i_sectors[12]中的间接块

（4）一级间接索引块表本身的扇区地址

您看，回收一个 inode 还是有些工作量的，本节咱们在 inode.c 中添加两个函数，依次回收以上罗列的相关资源，见代码 14-36。

代码 14-36　（project/c14/h/fs/inode.c）

```
…略
146    /* 将硬盘分区 part 上的 inode 清空 */
147    void inode_delete(struct partition* part, uint32_t inode_no, void* io_buf) {
148    ASSERT(inode_no < 4096);
149    struct inode_position inode_pos;
150    inode_locate(part, inode_no, &inode_pos);
       // inode 位置信息会存入 inode_pos
151    ASSERT(inode_pos.sec_lba <= (part->start_lba + part->sec_cnt));
152
153    char* inode_buf = (char*)io_buf;
154    if (inode_pos.two_sec) {   // inode 跨扇区，读入 2 个扇区
155       /* 将原硬盘上的内容先读出来 */
156       ide_read(part->my_disk, inode_pos.sec_lba, inode_buf, 2);
157       /* 将 inode_buf 清 0 */
158       memset((inode_buf + inode_pos.off_size), 0, sizeof(struct inode));
159       /* 用清 0 的内存数据覆盖磁盘 */
160       ide_write(part->my_disk, inode_pos.sec_lba, inode_buf, 2);
161    } else {    // 未跨扇区，只读入 1 个扇区就好
162       /* 将原硬盘上的内容先读出来 */
163       ide_read(part->my_disk, inode_pos.sec_lba, inode_buf, 1);
164       /* 将 inode_buf 清 0 */
165       memset((inode_buf + inode_pos.off_size), 0, sizeof(struct inode));
166       /* 用清 0 的内存数据覆盖磁盘 */
167       ide_write(part->my_disk, inode_pos.sec_lba, inode_buf, 1);
168    }
169 }
170
171    /* 回收 inode 的数据块和 inode 本身 */
172    void inode_release(struct partition* part, uint32_t inode_no) {
173    struct inode* inode_to_del = inode_open(part, inode_no);
174    ASSERT(inode_to_del->i_no == inode_no);
175
176    /* 1 回收 inode 占用的所有块 */
177    uint8_t block_idx = 0, block_cnt = 12;
178    uint32_t block_bitmap_idx;
179    uint32_t all_blocks[140] = {0};       //12 个直接块+128 个间接块
180
181    /* a 先将前 12 个直接块存入 all_blocks */
182    while (block_idx < 12) {
183       all_blocks[block_idx] = inode_to_del->i_sectors[block_idx];
184       block_idx++;
185    }
186
187    /* b 如果一级间接块表存在，将其 128 个间接块读到 all_blocks[12~],
       并释放一级间接块表所占的扇区 */
188    if (inode_to_del->i_sectors[12] != 0) {
189       ide_read(part->my_disk, inode_to_del->i_sectors[12],
          all_blocks + 12, 1);
190       block_cnt = 140;
191
192       /* 回收一级间接块表占用的扇区 */
193       block_bitmap_idx = \
          inode_to_del->i_sectors[12] - part->sb->data_start_lba;
194       ASSERT(block_bitmap_idx > 0);
195       bitmap_set(&part->block_bitmap, block_bitmap_idx, 0);
196       bitmap_sync(cur_part, block_bitmap_idx, BLOCK_BITMAP);
197    }
198
199    /* c inode 所有的块地址已经收集到 all_blocks 中，下面逐个回收 */
200    block_idx = 0;
```

```
201     while (block_idx < block_cnt) {
202         if (all_blocks[block_idx] != 0) {
203              block_bitmap_idx = 0;
204              block_bitmap_idx = all_blocks[block_idx] - \
                 part->sb->data_start_lba;
205              ASSERT(block_bitmap_idx > 0);
206              bitmap_set(&part->block_bitmap, block_bitmap_idx, 0);
207              bitmap_sync(cur_part, block_bitmap_idx, BLOCK_BITMAP);
208         }
209         block_idx++;
210     }
211
212 /*2 回收该 inode 所占用的 inode */
213     bitmap_set(&part->inode_bitmap, inode_no, 0);
214     bitmap_sync(cur_part, inode_no, INODE_BITMAP);
215
216     /******     以下 inode_delete 是调试用的    ******
217     * 此函数会在 inode_table 中将此 inode 清 0,
218     * 但实际上是不需要的, inode 分配是由 inode 位图控制的,
219     * 硬盘上的数据不需要清 0, 可以直接覆盖*/
220     void* io_buf = sys_malloc(1024);
221     inode_delete(part, inode_no, io_buf);
222     sys_free(io_buf);
223     /**********************************************/
224
225     inode_close(inode_to_del);
226 }
…略
```

代码看上去有点长，但实际上最上面的函数 inode_delete 是可有可无的，它是为了帮助调试而添加的，并不是必须的功能。

inode 的使用情况是由 inode 位图来控制的，从 inode 位图中把 inode 分配出去后，无论该 inode 中原来是否有数据，创建后一律会被新数据覆盖，因此在回收 inode 时只要在 inode 位图中的相应位置 0 就可以了，没必要在 inode_table 中真正擦除该 inode，就像删除文件时不需要真正把文件数据块中的数据擦除一样。

inode_delete 接受 3 个参数，分区 part、inode 编号 inode_no 及缓冲区 io_buf，功能是把 inode 从 inode_table 中擦除，也就是在 inode_table 中把该 inode 对应的空间清 0。

函数开头定义的变量 inode_pos 用于存储 inode 的位置信息，下面将它作为参数调用函数 inode_locate 以定位编号为 inode_no 的 inode，之后 inode_pos 中便有了该 inode 的“坐标”。

之后开始判断该 inode 是否跨扇区，并针对这两种情况分别处理。这两种情况的区别是从硬盘上读入 1 个扇区，还是读入 2 个扇区到 io_buf，共性把该 inode 从硬盘所在扇区读入到 io_buf 后，调用 memset 函数将 io_buf 中该 inode 所在的内存空间清 0，然后将该 io_buf 重新写回到硬盘，从而实现了将 inode 擦除的目的。

下面是函数 inode_release，它接受 2 个参数，分区 part 和 inode 编号 inode_no，功能是回收 inode，这包括 inode 中的数据块和 inode 本身在 inode 位图中的 bit。

由于文件操作的套路大伙儿已经熟悉了，我就不细说了，代码第 177～185 行是将直接块收集到 all_blocks。第 188～190 行判断，如果一级间接块表存在，将表中 128 个间接块读到 all_blocks 中第 12 块以后的其余空间中。一级间接块表本身占据 1 扇区，因此在第 193～196 行将该扇区回收。

执行到第 200 行时，该 inode 中所有的块地址已经被收集到 all_blocks 中，下面通过 while 循环把块逐个回收，核心操作是调用 bitmap_set 将内存中位图的 block_bitmap_idx 位置为 0，再调用 bitmap_sync 将内存中的位图同步到硬盘。关于块回收工作，这里有一点说明。如果该 inode 是普通文件，不需要遍历 140 个块，因为文件的数据是一个整体，存储数据时是把数据挨个、连续存放在块中的，相当于以 all_blocks[0]～all_blocks[139]的顺序依次写入，不会出现中间某个块地址为空（0）的情况，因此在 while 循环中可以把块地址为 0 作为结束条件。但如果该 inode 是目录的话，目录中的数据是目录项，它们都是单独的个体，每一个目录项都可以单独操作。不久我们就会了解，在删除文件中有一项工作就是擦除目录项，倘若该目录项单独占用 1 个块，为节约块资源，在删除此类目录项时我们会回收目录项所占的块（除了根目录中只剩下一个块的情况），也就是该块地址会被置为 0。该块很少是最后一个可用块，大部分情况下是位于 140

个块的中间某个块，因此对于目录要完整遍历 140 个块。举个例子，假设目录 a 中我们顺序创建了 90 个文件，恰好是每 30 个文件占 1 个块，前 30 个文件占用块 a0，即 i_sector[0]=a0，中间 30 个文件占用块 a1，即 i_sector[1]=a1，后 30 个文件占用块 a2，即 i_sector[2]=a2，这时我们先把中间 30 个文件都删除了，原本这 30 个文件所占的块就要被回收，因此出现 i_sector[1]=0 的情况。为实现简单，这里就不单独判断 inode 是目录，还是普通文件了，一律按最大块数 140 遍历。

接下来第 213～214 行是回收该 inode 在 inode 位图中所占用的位，依然通过 bitmap_set 和 bitmap_sync 两步完成。

到此我们回收了 inode 位图中的位、inode 涉及到的块，前面说过了，inode 本身所在的空间没必要真正擦除，因此 inode 回收工作就结束了。但为了在功能验证时让大伙儿看得更清楚，下面调用了 inode_delete 函数把该 inode 在 inode_table 擦除。在第 220 行，我们申请了 1024 字节的 io_buf 作为 inode_delete 的参数，理由是函数 inode_delete 中根据 inode 是否跨扇区的情况有可能要读入 2 个扇区。

最后再调用 inode_close(inode_to_del)将 inode 关闭。

14.10.2　删除目录项

文件名是以目录项的形式存在的，删除文件必须在目录中将其目录项擦除。下面看一下删除目录项相关的工作。

（1）在文件所在的目录中擦除该文件的目录项，使其为 0。

（2）根目录是必须存在的，它是文件读写的根基，不应该被清空，它至少要保留 1 个块。如果目录项独占 1 个块，并且该块不是根目录最后一个块的话，将其回收。

（3）目录 inode 的 i_size 是目录项大小的总和，因此还要将 i_size 减去一个目录项的单位大小。

（4）目录 inode 改变后，要同步到硬盘。

下面我们在 dir.c 中增加函数 delete_dir_entry 完成这项工作，见代码 14-37。

代码 14-37　（project/c14/h/fs/dir.c）

```
…略
214 /* 把分区 part 目录 pdir 中编号为 inode_no 的目录项删除 */
215 bool delete_dir_entry(struct partition* part, \
    struct dir* pdir, uint32_t inode_no, void* io_buf) {
216    struct inode* dir_inode = pdir->inode;
217    uint32_t block_idx = 0, all_blocks[140] = {0};
218    /* 收集目录全部块地址 */
219    while (block_idx < 12) {
220       all_blocks[block_idx] = dir_inode->i_sectors[block_idx];
221       block_idx++;
222    }
223    if (dir_inode->i_sectors[12]) {
224       ide_read(part->my_disk, dir_inode->i_sectors[12], all_blocks + 12, 1);
225    }
226
227    /* 目录项在存储时保证不会跨扇区 */
228    uint32_t dir_entry_size = part->sb->dir_entry_size;
229    uint32_t dir_entrys_per_sec = (SECTOR_SIZE / dir_entry_size);
   // 每扇区最大的目录项数目
230    struct dir_entry* dir_e = (struct dir_entry*)io_buf;
231    struct dir_entry* dir_entry_found = NULL;
232    uint8_t dir_entry_idx, dir_entry_cnt;
233    bool is_dir_first_block = false;    // 目录的第 1 个块
234
235    /* 遍历所有块，寻找目录项 */
236    block_idx = 0;
237    while (block_idx < 140) {
238       is_dir_first_block = false;
239       if (all_blocks[block_idx] == 0) {
240          block_idx++;
241          continue;
242       }
```

```
243        dir_entry_idx = dir_entry_cnt = 0;
244        memset(io_buf, 0, SECTOR_SIZE);
245        /* 读取扇区，获得目录项 */
246        ide_read(part->my_disk, all_blocks[block_idx], io_buf, 1);
247
248        /* 遍历所有的目录项，
           统计该扇区的目录项数量及是否有待删除的目录项 */
249        while (dir_entry_idx < dir_entrys_per_sec) {
250            if ((dir_e + dir_entry_idx)->f_type != FT_UNKNOWN) {
251                if (!strcmp((dir_e + dir_entry_idx)->filename, ".")) {
252                    is_dir_first_block = true;
253                } else if (strcmp((dir_e + dir_entry_idx)->filename, ".") &&
254                    strcmp((dir_e + dir_entry_idx)->filename, "..")) {
255                    dir_entry_cnt++;
                   // 统计此扇区内的目录项个数，用来判断删除目录项后是否回收该扇区
256                    if ((dir_e + dir_entry_idx)->i_no == inode_no) {
                   // 如果找到此 i 结点，就将其记录在 dir_entry_found
257                        ASSERT(dir_entry_found == NULL);
               // 确保目录中只有一个编号为 inode_no 的 inode
               // 找到一次后 dir_entry_found 就不再是 NULL
258                        dir_entry_found = dir_e + dir_entry_idx;
259                        /* 找到后也继续遍历，统计总共的目录项数 */
260                    }
261                }
262            }
263            dir_entry_idx++;
264        }
265
266        /* 若此扇区未找到该目录项，继续在下个扇区中找 */
267        if (dir_entry_found == NULL) {
268            block_idx++;
269            continue;
270        }
271
272     /* 在此扇区中找到目录项后，
       清除该目录项并判断是否回收扇区，随后退出循环直接返回 */
273        ASSERT(dir_entry_cnt >= 1);
274     /* 除目录第 1 个扇区外，若该扇区上只有该目录项自己，
       则将整个扇区回收 */
275        if (dir_entry_cnt == 1 && !is_dir_first_block) {
276            /* a 在块位图中回收该块 */
277            uint32_t block_bitmap_idx = \
           all_blocks[block_idx] - part->sb->data_start_lba;
278            bitmap_set(&part->block_bitmap, block_bitmap_idx, 0);
279            bitmap_sync(cur_part, block_bitmap_idx, BLOCK_BITMAP);
280
281            /* b 将块地址从数组 i_sectors 或索引表中去掉 */
282            if (block_idx < 12) {
283                dir_inode->i_sectors[block_idx] = 0;
284             } else {    // 在一级间接索引表中擦除该间接块地址
285             /*先判断一级间接索引表中间接块的数量，
             如果仅有这 1 个间接块，连同间接索引表所在的块一同回收 */
286                uint32_t indirect_blocks = 0;
287                uint32_t indirect_block_idx = 12;
288                while (indirect_block_idx < 140) {
289                   if (all_blocks[indirect_block_idx] != 0) {
290                      indirect_blocks++;
291                   }
292                }
293                ASSERT(indirect_blocks >= 1);  // 包括当前间接块
294
295                if (indirect_blocks > 1) {
             // 间接索引表中还包括其他间接块，仅在索引表中擦除当前这个间接块地址
296                   all_blocks[block_idx] = 0;
297                   ide_write(part->my_disk, \
             dir_inode->i_sectors[12], all_blocks + 12, 1);
298                } else {    // 间接索引表中就当前这 1 个间接块
    // 直接把间接索引表所在的块回收，然后擦除间接索引表块地址
299                   /* 回收间接索引表所在的块 */
300                   block_bitmap_idx = \
             dir_inode->i_sectors[12] - part->sb->data_start_lba;
301                   bitmap_set(&part->block_bitmap, block_bitmap_idx, 0);
```

```
302                    bitmap_sync(cur_part, block_bitmap_idx, BLOCK_BITMAP);
303
304                    /* 将间接索引表地址清 0 */
305                    dir_inode->i_sectors[12] = 0;
306                 }
307              }
308         } else { // 仅将该目录项清空
309              memset(dir_entry_found, 0, dir_entry_size);
310              ide_write(part->my_disk, all_blocks[block_idx], io_buf, 1);
311         }
312
313      /* 更新 i 结点信息并同步到硬盘 */
314         ASSERT(dir_inode->i_size >= dir_entry_size);
315         dir_inode->i_size -= dir_entry_size;
316         memset(io_buf, 0, SECTOR_SIZE * 2);
317         inode_sync(part, dir_inode, io_buf);
318
319         return true;
320      }
321/* 所有块中未找到则返回 false，出现这种情况应该是 serarch_file 出错了 */
322      return false;
323 }
```

delete_dir_entry 接受 4 个参数，分区 part、目录 pdir、inode 编号 inode_no、缓冲区 io_buf，功能是把分区 part 目录 pdir 中编号为 inode_no 的目录项删除。

代码第 218～225 行收集目录的 inode 占用的所有块，接下来在第 235～结束，开始遍历所有块，在其中查找目录项。

变量 is_dir_first_block 表示当前的块（待删除的目录项所在的块）是否是目录中最初的那个块，若在目录中创建很多文件或子目录后，目录会扩展到多个块中。根目录最初的块是在格式化分区时创建的，目录“.”和“..”都在这个块中，因此目录项的名称若为“.”，该块便是目录的最初的块。不禁您要问了，判断它有什么用呢？原因是：当删除一个目录项时，若该目录项所在的块上没有其他目录项了，或者是除了“.”和“..”之外没有其他目录项，我们就将该块回收了，否则空闲着一个块不是浪费吗。

第 243～246 行读取目录的块，获取该块中的目录项。接着在第 248～264 行开始遍历该块，寻找待删除的目录项。目录项成员 f_type 的值只要不是 FT_UNKNOWN 就表示该目录项有意义，第 251 行，若判断出目录项的 filename 为”.”，就表示当前的块是目录最初的块，因此将变量 is_dir_first_block 置为 true。否则统计目录中除“.”和“..”之外的所有目录项，目录项总数存储在变量 dir_entry_cnt 中。统计目录项个数的目的是判断删除目录项后是否回收该块，理由是若 dir_entry_cnt 为 1，就表示该目录项独占一个块，后面会根据是否为目录的最初块判断是否将该块回收。

第 256 行判断，如果目录项与待删除的 inode 编号相同，这说明找到了，用指针变量 dir_entry_found 记录其在 io_buf 中的地址，即 dir_e + dir_entry_idx，后面会用它来擦除目录项。

第 267 行判断，若此块中没找到该目录项，继续下一轮循环查找。如果找到了，就准备擦除该目录项并判断是否回收目录项所在的块。

第 275 行，如果当前块中目录项个数 dir_entry_cnt 为 1，并且当前块并不是根目录最初的那个块，那么就不需要擦除目录项，把当前块直接回收一了百了。工作分为两部分，先在第 277～279 行在块位图中回收当前块，然后将块地址从数组 i_sectors 或索引表中去掉。这部分工作涉及到索引表中的间接块，因此有可能涉及到索引表所在块的回收。第 282 行判断，当前块若是直接块，就在 i_sectors 数组中将相应位置为 0，这是最简单的情况。否则当前块是间接块，先判断一级间接索引表统计间接块的数量，对应的代码是第 286～293 行。第 295 行判断如果表中有多个间接块，间接索引表不能回收，就在第 296～297 行将该间接块地址清 0，同步到硬盘上的一级间接块索引表中。

第 298 行，如果表中仅有这 1 个间接块，就在第 300～302 行，把间接索引表所在的块一同回收，随后把 i_sector[12]置为 0，擦除间接块索引表地址。

若 275 行判断不成立，也就是如果当前块中的目录项个数为多个或者当前块是目录的最初块，此种情况不能将块回收，直接执行第 309～310 行，擦除目录项，然后同步到硬盘。

最后在第 314～317 行，将目录的 inode->i_size 减少一个目录项大小，然后同步 inode 到硬盘，返回 true。如果目录所有的块中都未找到待删除的目录项，就返回 false。

好啦，本节到这就结束了，咱们离文件删除的目标不远了，大伙儿加油。

14.10.3 实现 sys_unlink 与功能验证

前面基础搭的差不多了，终于到了删除文件的环节，Linux 下删除文件是用的 unlink 系统调用，其原型是“int unlink(const char *pathname)”，成功删除文件返回 0，否则返回−1。咱们还是先实现其内核部分——sys_unlink，见代码 14-38。

代码 14-38 （project/c14/h/fs/fs.c）

```
…略
450 /* 删除文件（非目录），成功返回 0，失败返回-1 */
451 int32_t sys_unlink(const char* pathname) {
452     ASSERT(strlen(pathname) < MAX_PATH_LEN);
453
454     /* 先检查待删除的文件是否存在 */
455     struct path_search_record searched_record;
456     memset(&searched_record, 0, sizeof(struct path_search_record));
457     int inode_no = search_file(pathname, &searched_record);
458     ASSERT(inode_no != 0);
459     if (inode_no == -1) {
460         printk("file %s not found!\n", pathname);
461         dir_close(searched_record.parent_dir);
462         return -1;
463     }
464     if (searched_record.file_type == FT_DIRECTORY) {
465         printk("can`t delete a direcotry with unlink(), \
        use rmdir() to instead\n");
466         dir_close(searched_record.parent_dir);
467         return -1;
468     }
469
470     /* 检查是否在已打开文件列表（文件表）中 */
471     uint32_t file_idx = 0;
472     while (file_idx < MAX_FILE_OPEN) {
473         if (file_table[file_idx].fd_inode != NULL &&\
    (uint32_t)inode_no == file_table[file_idx].fd_inode->i_no) {
474             break;
475         }
476         file_idx++;
477     }
478     if (file_idx < MAX_FILE_OPEN) {
479         dir_close(searched_record.parent_dir);
480         printk("file %s is in use, not allow to delete!\n", pathname);
481         return -1;
482     }
483     ASSERT(file_idx == MAX_FILE_OPEN);
484
485     /* 为 delete_dir_entry 申请缓冲区 */
486     void* io_buf = sys_malloc(SECTOR_SIZE + SECTOR_SIZE);
487     if (io_buf == NULL) {
488         dir_close(searched_record.parent_dir);
489         printk("sys_unlink: malloc for io_buf failed\n");
490         return -1;
491     }
492
493     struct dir* parent_dir = searched_record.parent_dir;
494     delete_dir_entry(cur_part, parent_dir, inode_no, io_buf);
495     inode_release(cur_part, inode_no);
496     sys_free(io_buf);
497     dir_close(searched_record.parent_dir);
498     return 0;   // 成功删除文件
499 }
…略
```

函数 sys_unlink 接受 1 个参数，文件绝对路径名 pathname，删除文件（非目录），成功返回 0，失败

返回−1。

因为代码中大部分方法及套路都是类似的，所以下面的介绍就不细致到各行了。第 455～463 行声明 path_search_record 结构并调用 search_file 检查文件 pathname 是否存在，如果不存在，则输出提示并返回−1。若只存在同名的目录，在第 464～467 行提示不能用 unlink 删除目录，只能用 rmdir 函数（将来实现）并返回−1。

第 470～477 行在文件表中检索待删除的文件，如果文件在文件表中存在，这说明该文件正被打开，不能删除。第 478～481 行判断如果文件已位于文件表中被打开了，输出提示该文件正处于使用中，不允许删除，返回−1。

第 485～491 行为即将调用的 delete_dir_entry 函数申请缓冲区，这里为其申请的缓冲区大小是两个扇区。

下面调用函数 delete_dir_entry 删除目录项，调用 inode_release 释放 inode，调用 dir_close 关闭 pathname 所在的目录后，返回 0，函数结束。

sys_unlink 完成了，下面进行功能测试，还是在 main.c 中添加测试代码，如代码 14-39 所示。

代码 14-39　（project/c14/h/kernel/main.c）

```
…略
19 int main(void) {
20     put_str("I am kernel\n");
21     init_all();
22     process_execute(u_prog_a, "u_prog_a");
23     process_execute(u_prog_b, "u_prog_b");
24     thread_start("k_thread_a", 31, k_thread_a, "I am thread_a");
25     thread_start("k_thread_b", 31, k_thread_b, "I am thread_b");
26     printf("/file1 delete %s!\n", sys_unlink("/file1") == 0 ? "done" : "fail");
27     while(1);
28     return 0;
29 }
…略
```

第 26 行代码是添加的 sys_unlink 调用，如果删除文件/file1 成功就输出“/file1 delete done！\n”，否则输出“/file1 delete fail！\n”。在程序执行之前，为方便对比文件删除的效果，咱们先查看下文件相关的元信息。图 14-18 中显示了块位图扇区地址是 0x41，inode 位图扇区地址是 0x49，inode_table 地址是 0x4a，根目录扇区地址是 0x2aa。咱们依次查看下它们的状态，如图 14-29 所示。

```
[work@localhost h]$ sh ~/tool/calculator.sh 0x41*512 d
33280
[work@localhost h]$ sh ~/tool/xxd.sh ~/my_workspace/bochs/hd80M.img 33280 512
0008200: 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0008210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00083f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[work@localhost h]$ sh ~/tool/calculator.sh 0x49*512 d
37376
[work@localhost h]$ sh ~/tool/xxd.sh ~/my_workspace/bochs/hd80M.img 37376 512
0009200: 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0009210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00093f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[work@localhost h]$ sh ~/tool/calculator.sh 0x4a*512 d
37888
[work@localhost h]$ sh ~/tool/xxd.sh ~/my_workspace/bochs/hd80M.img 37888 512
0009400: 00 00 00 00 48 00 00 00 00 00 00 00 00 00 00 00  ....H...........
0009410: AA 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0009420: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0009430: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0009440: 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00  ................
0009450: 18 00 00 00 00 00 00 00 00 00 00 00 AB 02 00 00  ................
0009460: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00095f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[work@localhost h]$ sh ~/tool/calculator.sh 0x2aa*512 d
349184
[work@localhost h]$ sh ~/tool/xxd.sh ~/my_workspace/bochs/hd80M.img 349184 512
0055400: 2E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0055410: 00 00 00 00 02 00 00 00 2E 2E 00 00 00 00 00 00  ................
0055420: 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00  ................
0055430: 66 69 6C 65 31 00 00 00 00 00 00 00 00 00 00 00  file1...........
0055440: 01 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00  ................
0055450: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00555f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[work@localhost h]$
```

块位图

inode 位图

inode table

根目录

▲图 14−29　文件元信息（删除文件前）

块位图中的值是 3，这说明分配了 2 个块，一个分配给了根目录的 i_sectors[0]，另一个分配给了/file1 的 i_sectors[0]。

inode 位图中的值也是 3，同样是分别分配给了根目录和/file1 的 inode。

inode_table 中画框的部分是第 2 个 inode 以后的内容，具体字段的值之前有介绍过，不再重复说明。

根目录中的框框是/file1 的目录项。

下面执行程序，运行结果如图 14-30 所示。

```
Bochs x86 emulator, http://bochs.sourceforge.net/
      SECTORS: 163296
      CAPACITY: 79MB

   all partition info
   sdb1 start_lba:0x3F, sec_cnt:0x7DC1
   sdb5 start_lba:0x7E3F, sec_cnt:0x46A1
   sdb6 start_lba:0xC51F, sec_cnt:0x6231
   sdb7 start_lba:0x1278F, sec_cnt:0x3AD1
   sdb8 start_lba:0x1629F, sec_cnt:0x75E1
   sdb9 start_lba:0x1D8BF, sec_cnt:0xA521
ide_init done
searching filesystem......
sdb1 has filesystem
sdb5 has filesystem
sdb6 has filesystem
sdb7 has filesystem
sdb8 has filesystem
sdb9 has filesystem
mount sdb1 done!
 prog_a malloc addr:0x804800C,0x804810C,0x804820C
 prog_b malloc addr:0x804800C,0x804810C,0x804820C
 thread_a malloc addr:0xC013900C,C013910C,C013920C
 thread_b malloc addr:0xC013930C,C013940C,C013950C
/file1 delete done!
IPS: 26.546M    NUM  CAPS  SCRL  HD:0-M  HD:0-S
```

▲图 14-30　bochs 运行结果

图中最下行输出“/file1 delete done!\n”，下面验证下文件元信息，如图 14-31 所示。

```
[work@localhost h]$ sh ~/tool/xxd.sh ~/my_workspace/bochs/hd80M.img 33280 512
0008200: 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0008210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00083f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[work@localhost h]$ sh ~/tool/xxd.sh ~/my_workspace/bochs/hd80M.img 37376 512
0009200: 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0009210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00093f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[work@localhost h]$ sh ~/tool/xxd.sh ~/my_workspace/bochs/hd80M.img 37888 512
0009400: 00 00 00 00 30 00 00 00 00 00 00 00 00 00 00 00  ....0...........
0009410: AA 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0009420: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00095f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[work@localhost h]$ sh ~/tool/xxd.sh ~/my_workspace/bochs/hd80M.img 349184 512
0055400: 2E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0055410: 00 00 00 00 02 00 00 00 2E 2E 00 00 00 00 00 00  ................
0055420: 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00  ................
0055430: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00555f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[work@localhost h]$ _
```

块位图

inode 位图

inode table

根目录

▲图 14-31　文件元信息（删除文件后）

和图 14-29 对比，图 14-31 中，块位图和 inode 位图都由 3 变成了 1，这说明 file1 所占的块被回收，inode 也被回收了。inode_table 和根目录中也没有相应的信息了，符合预期。顺便提一句，inode_table 中/file1 的 inode 是被函数 inode_delete 擦除的，其实没必要擦除该 inode，完全是为了此处的调试。

14.11 创建目录

本节开始，咱们让文件系统支持目录，其实目录的实现并不难，难的是如何使文件系统更加高效，这也恰恰是 Linux 的魅力所在——高效率！咱们还是本着学习的精神，效率、优化是以后的事，咱们先把目录实现从无到有。

14.11.1　实现 sys_mkdir 创建目录

Linux 用 mkdir 函数创建目录，还有一个同名的 mkdir 命令也用来创建目录，原理上都是一回事，只是一个是系统调用，另一个是利用此系统调用实现的可执行程序。

先看下 mkdir 的原型：“int mkdir(const char *pathname, mode_t mode)”，其中，pathname 是待创建的目录名，mode 是所创建目录的权限，成功返回 0，失败返回−1。如果是头一次接触 Linux 的同学，有可能不太了解文件的“权限”，这个权限是指用户、组内成员、其他成员对目录的读、写、执行能力的设置，不清楚也没关系，因为权限管理涉及到用户管理等周边功能，工作量有点庞大，而且这些是有关安全方面的内容，已不属于操作系统基本原理的范畴，违背了本书的初心，最要命的是，小弟我能力实在有限，哈哈，惭愧。所以咱们的 mkdir 就不支持权限参数 mode 了。

咳咳，下面说正事，创建目录所涉及的工作包括。

（1）确认待创建的新目录在文件系统上不存在。

（2）为新目录创建 inode。

（3）为新目录分配 1 个块存储该目录中的目录项。

（4）在新目录中创建两个目录项“.”和“..”，这是每个目录都必须存在的两个目录项。

（5）在新目录的父目录中添加新目录的目录项。

（6）将以上资源的变更同步到硬盘。

下面我们在 mkdir 的内核部分——sys_mkdir 中实现以上功能，见代码 14-40。

代码 14-40　（project/c14/i/fs/fs.c）

```
…略
501 /* 创建目录 pathname，成功返回 0，失败返回-1 */
502 int32_t sys_mkdir(const char* pathname) {
503     uint8_t rollback_step = 0;   // 用于操作失败时回滚各资源状态
504     void* io_buf = sys_malloc(SECTOR_SIZE * 2);
505     if (io_buf == NULL) {
506         printk("sys_mkdir: sys_malloc for io_buf failed\n");
507         return -1;
508     }
509
510     struct path_search_record searched_record;
511     memset(&searched_record, 0, sizeof(struct path_search_record));
512     int inode_no = -1;
513     inode_no = search_file(pathname, &searched_record);
514     if (inode_no != -1) {    // 如果找到了同名目录或文件，失败返回
515         printk("sys_mkdir: file or directory %s exist!\n", pathname);
516         rollback_step = 1;
517         goto rollback;
518     } else {
      // 若未找到，也要判断是在最终目录没找到，还是某个中间目录不存在
519         uint32_t pathname_depth = path_depth_cnt((char*)pathname);
520         uint32_t path_searched_depth =\
      path_depth_cnt(searched_record.searched_path);
521         /* 先判断是否把 pathname 的各层目录都访问到了，
            即是否在某个中间目录就失败了 */
522         if (pathname_depth != path_searched_depth) {
          // 说明并没有访问到全部的路径，某个中间目录是不存在的
523             printk("sys_mkdir: cannot access %s: Not a directory,
        subpath %s is`t exist\n", pathname,
        searched_record.searched_path);
524             rollback_step = 1;
525             goto rollback;
526         }
527     }
528
529     struct dir* parent_dir = searched_record.parent_dir;
530     /* 目录名称后可能会有字符'/'，
        所以最好直接用 searched_record.searched_path，无'/' */
531     char* dirname = strrchr(searched_record.searched_path, '/') + 1;
532
```

```
533     inode_no = inode_bitmap_alloc(cur_part);
534     if (inode_no == -1) {
535         printk("sys_mkdir: allocate inode failed\n");
536         rollback_step = 1;
537         goto rollback;
538     }
539
540     struct inode new_dir_inode;
541     inode_init(inode_no, &new_dir_inode);  // 初始化i结点
542
543     uint32_t block_bitmap_idx = 0;
        // 用来记录block对应于block_bitmap中的索引
544     int32_t block_lba = -1;
545     /* 为目录分配一个块，用来写入目录.和.. */
546     block_lba = block_bitmap_alloc(cur_part);
547     if (block_lba == -1) {
548         printk("sys_mkdir: block_bitmap_alloc
            for create directory failed\n");
549         rollback_step = 2;
550         goto rollback;
551     }
552     new_dir_inode.i_sectors[0] = block_lba;
553     /* 每分配一个块就将位图同步到硬盘 */
554     block_bitmap_idx = block_lba - cur_part->sb->data_start_lba;
555     ASSERT(block_bitmap_idx != 0);
556     bitmap_sync(cur_part, block_bitmap_idx, BLOCK_BITMAP);
557
558     /* 将当前目录的目录项'.'和'..'写入目录 */
559     memset(io_buf, 0, SECTOR_SIZE * 2);   // 清空io_buf
560     struct dir_entry* p_de = (struct dir_entry*)io_buf;
561
562     /* 初始化当前目录"." */
563     memcpy(p_de->filename, ".", 1);
564     p_de->i_no = inode_no ;
565     p_de->f_type = FT_DIRECTORY;
566
567     p_de++;
568     /* 初始化当前目录".." */
569     memcpy(p_de->filename, "..", 2);
570     p_de->i_no = parent_dir->inode->i_no;
571     p_de->f_type = FT_DIRECTORY;
572     ide_write(cur_part->my_disk, new_dir_inode.i_sectors[0], io_buf, 1);
573
574     new_dir_inode.i_size = 2 * cur_part->sb->dir_entry_size;
575
576     /* 在父目录中添加自己的目录项 */
577     struct dir_entry new_dir_entry;
578     memset(&new_dir_entry, 0, sizeof(struct dir_entry));
579     create_dir_entry(dirname, inode_no, \
                         FT_DIRECTORY, &new_dir_entry);
580     memset(io_buf, 0, SECTOR_SIZE * 2);   // 清空io_buf
581     if (!sync_dir_entry(parent_dir, &new_dir_entry, io_buf)) {
     // sync_dir_entry中将block_bitmap通过bitmap_sync同步到硬盘
582         printk("sys_mkdir: sync_dir_entry to disk failed!\n");
583         rollback_step = 2;
584         goto rollback;
585     }
586
587     /* 父目录的inode同步到硬盘 */
588     memset(io_buf, 0, SECTOR_SIZE * 2);
589     inode_sync(cur_part, parent_dir->inode, io_buf);
590
591     /* 将新创建目录的inode同步到硬盘 */
592     memset(io_buf, 0, SECTOR_SIZE * 2);
593     inode_sync(cur_part, &new_dir_inode, io_buf);
594
595     /* 将inode位图同步到硬盘 */
596     bitmap_sync(cur_part, inode_no, INODE_BITMAP);
597
598     sys_free(io_buf);
599
600     /* 关闭所创建目录的父目录 */
```

```
601     dir_close(searched_record.parent_dir);
602     return 0;
603
604 /*创建文件或目录需要创建相关的多个资源，
    若某步失败则会执行到下面的回滚步骤 */
605 rollback:          // 因为某步骤操作失败而回滚
606     switch (rollback_step) {
607        case 2:
608           bitmap_set(&cur_part->inode_bitmap, inode_no, 0);
              // 如果新文件的 inode 创建失败，之前位图中分配的 inode_no 也要恢复
609        case 1:
610           /* 关闭所创建目录的父目录 */
611           dir_close(searched_record.parent_dir);
612           break;
613     }
614     sys_free(io_buf);
615     return -1;
616 }
…略
```

函数 sys_mkdir 支持 1 个参数，路径名 pathname，功能是创建目录 pathname，成功返回 0，失败返回−1。

创建目录也是由多个步骤完成的，因此创建目录的工作是个事务，具有原子性，即要么所有步骤都完成，要么一个都不做，若其中某个步骤失败，必须将之前完成的操作回滚到之前的状态。在函数开头定义的 rollback_step 用于记录回滚的步骤。接下来早早申请了 2 扇区大小的缓冲区给 io_buf，后面很多操作都要用到它。

在创建目录之前要判断文件系统上是否已经有了同名的文件，无论是目录文件，还是普通文件，同一目录下不允许存在同名的文件，第 510～513 行调用 search_file 检索 pathname，如果找到同名文件 pathname，search_file 会返回其 inode 编号，否则会返回−1。第 514 行判断 search_file 的返回值，若不等于−1 表示同名文件已存在，输出提示后将 rollback_step 置为 1，跳转到 rollback 处执行回滚。大伙儿不妨移步到 604 行，这是回滚步骤，这里有 case 2 和 case 1，case1 是关闭待创建目录的父目录，case2 在此基础上多了恢复 inode 位图的操作，好啦，赶紧回来。

第 518 行，如果未找到同名文件，这也不能贸然创建目录，因为待创建的目录有可能并不是在最后一级目录中不存在，很可能是某个中间路径就不存在，比如创建目录“/a/b”，有可能 a 目录就不存在。对于中间目录不存在的情况，我们就像 Linux 一样，给出提示后拒绝创建目录，第 519～526 行代码就是处理这种情况，这里是用 pathname 的路径深度 pathname_depth 和已搜索过的路径深度 path_searched_depth 比较，若不相等（确切地说，若 pathname_depth 大于 path_searched_depth）则表示某个中间目录不存在，该“中间目录”是 searched_record.searched_path。

第 529 行用指针 parent_dir 指向被创建目录所在的父目录，第 531 行获取 pathname 的最后一级目录名，也就是最终创建的目录名，其中 strrchr 函数是在 searched_record.searched_path 中从后往前获取字符'/'第一次出现的地址（注意，是地址，不是字符下标），如创建目录“/a/b”，若按照第 531 行的处理，目录名 dirname 指向 b。

接着调用 inode_bitmap_alloc 在 inode 位图中分配 inode，如果返回值为−1，将 rollback_step 置为 1，跳转到 rollback 处执行回滚。

第 540～541 行为目录新建 1 个 new_dir_inode 并初始化。第 543～552 行为目录分配 1 个块并将块地址写入目录 inode 的 i_sectors[0]中，此块用来存储目录中的目录项。第 554～556 行将块位图同步到硬盘。

第 559～572 行在 io_buf 中新建目录项“.”和“..”并同步到硬盘，这样目录 pathname 中便有了两个目录项。第 574 行初始化目录的尺寸，即 new_dir_inode.i_size 等于 2 个目录项大小。

接下来要在父目录中添加自己的目录项，即在 parent_dir 中添加 dirname 的目录项，这是由第 577～584 行的代码完成的。其中 create_dir_entry 只是初始化目录项的内容到 new_dir_entry 中，下面调用的函数“sync_dir_entry(parent_dir, &new_dir_entry, io_buf”才是真正把 dirname 的目录项 new_dir_entry 写入父目录 parent_dir 中。

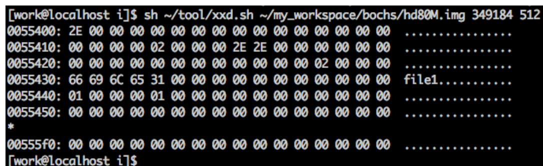
最后是一些元信息持久化工作，第 588～593 行同步父目录 inode 和新目录 inode 到硬盘，第 596 行同步 inode 位图到硬盘。

在第 598 行以后开始释放缓冲区 io_buf，关闭父目录 searched_record.parent_dir，随后返回 0，创建目

录成功。

14.11.2 创建目录功能验证

上节中我们已经把/file1 删除了，为了展示普通文件与目录的差别，我悄悄把/file1 又恢复了，因此下面的测试相当于穿越回了上节之前，根目录下仅有/file1 的情况，请大伙儿知晓。目前根目录下的情况如图 14-32 所示。



▲图 14-32　创建目录前根目录中的目录项

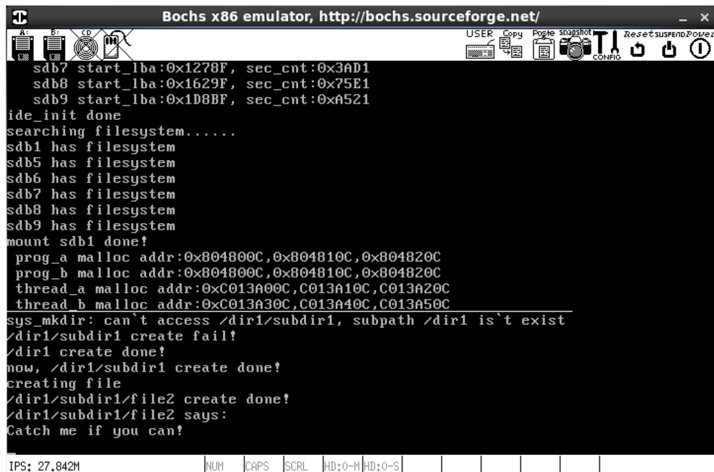
下面在 main.c 中加入创建目录的测试代码，如代码 14-41 所示。

代码 14-41　（project/c14/i/kernel/main.c）

```
…略
19 int main(void) {
…略
26     printf("/dir1/subdir1 create %s!\n", \
       sys_mkdir("/dir1/subdir1") == 0 ? "done" : "fail");
27     printf("/dir1 create %s!\n", sys_mkdir("/dir1") == 0 ? "done" : "fail");
28     printf("now, /dir1/subdir1 create %s!\n", \
       sys_mkdir("/dir1/subdir1") == 0 ? "done" : "fail");
29     int fd = sys_open("/dir1/subdir1/file2", O_CREAT|O_RDWR);
30     if (fd != -1) {
31         printf("/dir1/subdir1/file2 create done!\n");
32         sys_write(fd, "Catch me if you can!\n", 21);
33         sys_lseek(fd, 0, SEEK_SET);
34         char buf[32] = {0};
35         sys_read(fd, buf, 21);
36         printf("/dir1/subdir1/file2 says:\n%s", buf);
37         sys_close(fd);
38     }
39     while(1);
40     return 0;
41 }
…略
```

第 26～38 行是我们新加的测试代码，第 26 行想创建目录“/dir1/subdir1”。目前没有目录“/dir1”，因此直接创建“/dir1/subdir1”会失败。第 27 行先创建目录“/dir1”，第 28 行重新创建目录“/dir1/subdir1”。第 29 行在目录“/dir1/subdir1”下创建文件“file2”。第 32 行往文件“/dir1/subdir1/file2”中写数据“Catch me if you can!\n”。第 35～36 行将其读出并打印。

如图 14-33 所示是运行结果。



▲图 14-33　目录创建

图中横线以下的部分是本次测试代码相关的输出，功能符合预期，其中“sys_mkdir:can’t access

/dir1/subdir1, subpath /dir1 is't exist”是 sys_mkdir 函数的输出，提示父目录“/dir1”不存在。“creating file”是 sys_open 的输出，以后我们会把此类提示去掉，请您提前知晓。

下面看下相关文件在硬盘上的结果，先看下根目录的目录项，如图 14-34 所示。

```
[work@localhost i]$ sh ~/tool/xxd.sh ~/my_workspace/bochs/hd80M.img 349184 512
0055400: 2E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0055410: 00 00 00 00 02 00 00 00 2E 2E 00 00 00 00 00 00  ................
0055420: 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00  ................
0055430: 66 69 6C 65 31 00 00 00 00 00 00 00 00 00 00 00  file1...........
0055440: 01 00 00 00 01 00 00 00 64 69 72 31 00 00 00 00  ........dir1....
0055450: 00 00 00 00 00 00 00 00 02 00 00 00 02 00 00 00  ................
0055460: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00555f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[work@localhost i]$
```

▲图 14-34　创建目录后根目录中的目录项

对比图 14-32 创建 dir1 前的根目录，图 14-34 中多了目录 dir1，其中标下画线的部分是 dir1 的目录项。第一个下画线是目录项的 filename，第二个下画线标出的 02 是 inode 编号，第三个下画线标出的 02 是文件类型，即 FT_DIRECTORY，这说明 dir1 是目录。下面我们到 inode_table 中查看第 02 个 inode 的信息，如图 14-35 所示。

```
[work@localhost i]$ sh ~/tool/xxd.sh ~/my_workspace/bochs/hd80M.img 37888 512
0009400: 00 00 00 00 60 00 00 00 00 00 00 00 00 00 00 00  ....`...........
0009410: AA 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0009420: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0009430: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0009440: 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00  ................
0009450: 18 00 00 00 00 00 00 00 00 00 00 00 AB 02 00 00  ................
0009460: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
0009490: 00 00 00 00 00 00 00 00 02 00 00 00 48 00 00 00  ............H...
00094a0: 00 00 00 00 00 00 00 00 AC 02 00 00 00 00 00 00  ................
00094b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00094e0: 00 00 00 00 03 00 00 00 48 00 00 00 00 00 00 00  ........H.......
00094f0: 00 00 00 00 AD 02 00 00 00 00 00 00 00 00 00 00  ................
0009500: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
0009530: 04 00 00 00 15 00 00 00 00 00 00 00 00 00 00 00  ................
0009540: AE 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0009550: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00095f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[work@localhost i]$
```

▲图 14-35　inode_table 中的 inode

图中第一个框框中，上面的“02 00 00 00”是 dir1 的 i_no，即 inode 编号，下面的“AC 02 00 00”是 i_sectors[0]的值，这说明 dir1 中的目录项在扇区 0x2ac 处。在这两个数之间的三组 4 字节数据，分别是 i_size、i_open_cnts 和 write_deny。下面看地址 0x2ac 处的数据，如图 14-36 所示。

```
[work@localhost i]$ sh ~/tool/xxd.sh ~/my_workspace/bochs/hd80M.img 350208 512
0055800: 2E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0055810: 02 00 00 00 02 00 00 00 2E 2E 00 00 00 00 00 00  ................
0055820: 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00  ................
0055830: 73 75 62 64 69 72 31 00 00 00 00 00 00 00 00 00  subdir1.........
0055840: 03 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00  ................
0055850: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
00559f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[work@localhost i]$
```

▲图 14-36　“/dir1”中的目录项

图 14-36 中方框的内容分别是“.”和“..”的目录项。下面标下画线的部分是 subdir1 的目录项。第一个长下画线是目录项的 filename，值为 subdir1。第二个下画线的值是 03，这是 inode 编号，其值为 3，第三个下画线是文件类型，其值为 02，即 FT_DIRECTORY。在图 14-35 的第二个方框中的是 subdir1 的 inode 的部分信息，该方框中上面的是 inode 编号，下面的是 i_sectors[0]，其值为 0x2ad，这说明 subdir1 的目录项在扇区 0x2ad 处，继续追踪。

图 14-37 中的方框分别是“.”和“..”的目录项，下面下画线标出的是 file2 的目录项，其中的 04 是 inode 编号，01 是文件类型，即 FT_REGULAR，这说明 file2 是普通文件。在图 14-35 中第三个方框是 file2 的 inode 的部分信息，其 i_sector[0]的值为 0x2ae，继续追踪。

```
[work@localhost i]$ sh ~/tool/calculator.sh 0x2ad*512 d
350720
[work@localhost i]$ sh ~/tool/xxd.sh ~/my_workspace/bochs/hd80M.img 350720 512
0055a00: 2E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0055a10: 03 00 00 00 02 00 00 00 2E 2E 00 00 00 00 00 00  ................
0055a20: 00 00 00 00 00 00 00 00 02 00 00 00 02 00 00 00  ................
0055a30: 66 69 6C 65 32 00 00 00 00 00 00 00 00 00 00 00  file2...........
0055a40: 04 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00  ................
0055a50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
0055bf0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[work@localhost i]$
```

▲图 14-37 “/dir1/subdir1”的目录项

```
[work@localhost i]$ sh ~/tool/calculator.sh 0x2ae*512 d
351232
[work@localhost i]$ sh ~/tool/xxd.sh ~/my_workspace/bochs/hd80M.img 351232 512
0055c00: 43 61 74 63 68 20 6D 65 20 69 66 20 79 6F 75 20  Catch me if you
0055c10: 63 61 6E 21 0A 00 00 00 00 00 00 00 00 00 00 00  can!...........
0055c20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
0055df0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[work@localhost i]$
```

▲图 14-38 “/dir1/subdir1/file2”的内容

图 14-38 是新创建的文件“/dir1/subdir1/file2”的内容，真不容易，历经千辛万苦终于抓到了它。

好啦，本节搞定，大伙儿收摊了。

14.12 遍历目录

本节咱们要实现目录的读取，先从打开目录开始。

14.12.1 打开目录和关闭目录

遍历目录就是读取目录中所有的目录项，在遍历之前必须要先把目录打开，之后还需要把目录关闭。Linux 中分别用函数 opendir 和 closedir 完成目录打开和关闭，原型分别是：

```
DIR *opendir(const char *name)和 int closedir(DIR *dirp);
```

咱们模仿这两个接口实现自己的版本。还是先实现 opendir 和 closedir 的内核部分——sys_opendir 和 sys_closedir，它们定义在 fs.c 中，见代码 14-42。

代码 14-42 （project/c14/j/fs/fs.c）

```
...略
618    /* 目录打开成功后返回目录指针，失败返回 NULL */
619    struct dir* sys_opendir(const char* name) {
620    ASSERT(strlen(name) < MAX_PATH_LEN);
621    /* 如果是根目录'/'，直接返回&root_dir */
622    if (name[0] == '/' && (name[1] == 0 || name[0] == '.')) {
623        return &root_dir;
624    }
625
626    /* 先检查待打开的目录是否存在 */
627    struct path_search_record searched_record;
628    memset(&searched_record, 0, sizeof(struct path_search_record));
629    int inode_no = search_file(name, &searched_record);
630    struct dir* ret = NULL;
631    if (inode_no == -1) {    //如果找不到目录，提示不存在的路径
632        printk("In %s, sub path %s not exist\n", \
           name, searched_record.searched_path);
633    } else {
634        if (searched_record.file_type == FT_REGULAR) {
635            printk("%s is regular file!\n", name);
636        } else if (searched_record.file_type == FT_DIRECTORY) {
637            ret = dir_open(cur_part, inode_no);
638        }
639    }
640    dir_close(searched_record.parent_dir);
641    return ret;
642 }
```

```
643
644     /* 成功关闭目录 p_dir 返回 0，失败返回-1 */
645     int32_t sys_closedir(struct dir* dir) {
646     int32_t ret = -1;
647     if (dir != NULL) {
648         dir_close(dir);
649         ret = 0;
650     }
651     return ret;
652 }
...略
```

sys_opendir 接受一个参数 name，功能是打开目录 name，成功后返回目录指针，失败返回 NULL。

根目录的形式有："/"、"/."、"/.."，当然按理说"/./.."、"/../.."等都能够表示根目录，但毕竟实属"罕见"，因此暂不考虑它们。第 622 行判断打开的目录是否是根目录，如果是就直接把根目录地址&root_dir 返回，这里是简单处理"/."和"/.."的情况。

打开目录之前要确认目录存在，否则失败返回，因此第 627～629 行通过 search_file 在文件系统上查找目录 name。第 630 行用变量 ret 存储目录指针，默认为 NULL。若找不到 name，在第 631～632 行提示路径不存在。若找到了，第 634～635 行判断如果找到的是普通文件，就输出提示，否则若找到的确实是目录，就调用 dir_open 将 name 打开，目录指针存入 ret 中，随后在第 640 行调用 dir_close 关闭 name 的父目录，最后返回 ret 指针。

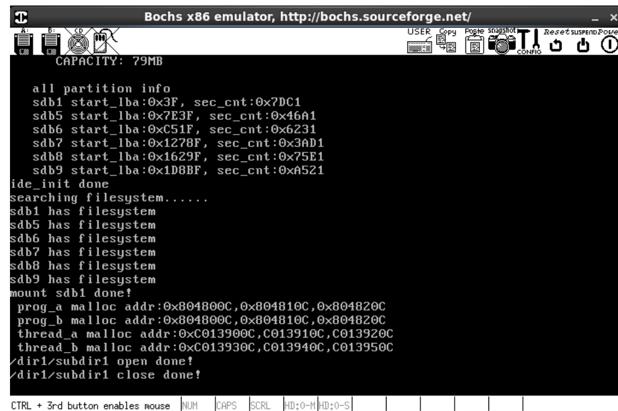
接下来是 sys_closedir 函数，接受一个参数目录指针 dir，功能是关闭目录 dir，成功返回 0，失败返回−1。它的实现太简单了，关闭目录的具体工作是调用 dir_close(dir)完成的。

我们在 main.c 中加入这两个函数的测试，见代码 14-43。

代码 14-43　（project/c14/j/kernel/main.c）

```
…略
19 int main(void) {
…略
26     struct dir* p_dir = sys_opendir("/dir1/subdir1");
27     if (p_dir) {
28         printf("/dir1/subdir1 open done!\n");
29         if (sys_closedir(p_dir) == 0) {
30             printf("/dir1/subdir1 close done!\n");
31         } else {
32             printf("/dir1/subdir1 close fail!\n");
33         }
34     } else {
35         printf("/dir1/subdir1 open fail!\n");
36     }
37     while(1);
38     return 0;
39 }
…略
```

测试代码很简单，运行结果如图 14-39 所示。



▲图 14−39　目录的打开与关闭

图中最后两行是打开及关闭目录的输出信息，符合预期。

本节内容不多，到这就结束了，下节再续。

14.12.2 读取 1 个目录项

前面说过啦，读取目录实际上就是读取目录中的目录项。在进行开发之前，咱们先了解下 Linux 读取目录的方法，具体如图 14-40 所示。

图 14-40 是遍历根目录的例子。第 6 行先声明了目录指针变量 p_dir，第 7 行声明了目录项指针变量 dir_e，在第 8 行用 opendir 函数打开根目录“/”，返回目录地址给 p_dir。然后再把 p_dir 作为 readdir 函数的参数，循环调用，每次返回一个目录项地址赋给 dir_e，直到返回值为空，即 NULL。然后在第 11 行输出目录项的名字，即 dir_e->d_name。

总结一下重点：readdir 每次返回目录的一个目录项地址，因此遍历目录需要循环调用 readdir。

咱们也本着同样的原则实现自己的 readdir。遍历目录不可能仅是 readdir 一个人的功劳，其核心是定义在 dir.c 中的函数 dir_read，见代码 14-44。

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <dirent.h>
 4
 5 int main() {
 6    DIR* p_dir = NULL;
 7    struct dirent* dir_e = NULL;
 8    p_dir = opendir("/");
 9    if (p_dir) {
10       while(dir_e = readdir(p_dir)) {
11          printf("%s\n", dir_e->d_name);
12       }
13    }
14    return 0;
15 }
```

▲图 14-40 Linux 中读取目录

代码 14-44 （project/c14/k/fs/dir.c）

```
…略
325 /*读取目录，成功返回 1 个目录项，失败返回 NULL */
326 struct dir_entry* dir_read(struct dir* dir) {
327     struct dir_entry* dir_e = (struct dir_entry*)dir->dir_buf;
328     struct inode* dir_inode = dir->inode;
329     uint32_t all_blocks[140] = {0}, block_cnt = 12;
330     uint32_t block_idx = 0, dir_entry_idx = 0;
331     while (block_idx < 12) {
332        all_blocks[block_idx] = dir_inode->i_sectors[block_idx];
333        block_idx++;
334     }
335     if (dir_inode->i_sectors[12] != 0) {      // 若含有一级间接块表
336        ide_read(cur_part->my_disk, dir_inode->i_sectors[12], \
            all_blocks + 12, 1);
337        block_cnt = 140;
338     }
339     block_idx = 0;
340
341     uint32_t cur_dir_entry_pos = 0;
        // 当前目录项的偏移，此项用来判断是否是之前已经返回过的目录项
342     uint32_t dir_entry_size = cur_part->sb->dir_entry_size;
343     uint32_t dir_entrys_per_sec = SECTOR_SIZE / dir_entry_size;
        // 1 扇区内可容纳的目录项个数
344     /* 在目录大小内遍历 */
345     while (dir->dir_pos < dir_inode->i_size) {
346        if (dir->dir_pos >= dir_inode->i_size) {
347            return NULL;
348        }
349        if (all_blocks[block_idx] == 0) {
           // 如果此块地址为 0，即空块，继续读出下一块
350            block_idx++;
351            continue;
352        }
353        memset(dir_e, 0, SECTOR_SIZE);
354        ide_read(cur_part->my_disk, all_blocks[block_idx], dir_e, 1);
355        dir_entry_idx = 0;
356        /* 遍历扇区内所有目录项 */
357        while (dir_entry_idx < dir_entrys_per_sec) {
358            if ((dir_e + dir_entry_idx)->f_type) {
     // 如果 f_type 不等于 0，即不等于 FT_UNKNOWN
359  /* 判断是不是最新的目录项，避免返回曾经已经返回过的目录项 */
360               if (cur_dir_entry_pos < dir->dir_pos) {
361                  cur_dir_entry_pos += dir_entry_size;
```

```
362                 dir_entry_idx++;
363                 continue;
364             }
365             ASSERT(cur_dir_entry_pos == dir->dir_pos);
366             dir->dir_pos += dir_entry_size;
            // 更新为新位置，即下一个返回的目录项地址
367             return dir_e + dir_entry_idx;
368         }
369         dir_entry_idx++;
370     }
371     block_idx++;
372   }
373   return NULL;
374 }
```

函数 dir_read 接受 1 个参数，目录指针 dir，功能是读取目录 dir，成功返回 1 个目录项，失败返回 NULL。

在目录结构中有个 512 字节的缓冲区 dir_buf，也许您之前不了解它是做什么的，现在告诉您，它的作用是存储目录项。首先在程序开头声明了目录项指针 dir_e，使其指向目录缓冲区 dir_buf。为读取目录，必然要知道目录 inode 所有的块地址，大伙儿早已经熟悉了我的套路，没错，在第 329～338 行将目录所有块地址收集到 all_blocks 中，并使块索引 block_idx 恢复为 0。

接下来遍历该目录所有的块，然后在每个块中遍历目录项。问题来了，既然 dir_read 也是只返回 1 个目录项，那如何知道该返回哪个目录项呢？到了目录成员 dir_pos“发威”的时候了，dir_pos 是目录的“游标”，作用同文件结构中的 fd_pos 一样，用于记录下一个读写对象的地址，dir_pos 用于指向目录中某个目录项的地址。我们每返回一个目录项后就使 dir_pos 的值增加一个目录项大小，这样就有可能知道该返回哪个目录项了。由于目录中的目录项是单独的个体，它们可以被单独删除，这样就会使块中存在“空洞”，也就是目录中的目录项不连续存储，将它们读入到内存缓冲区中，这些“空洞”也存在，我们并没有在内存中整理目录项使其连续，因为这样会导致效率更加低下，也没必要。所以仅凭 dir_pos 还不能满足要求，我们得知道哪些目录项已经被读取过了，在第 341 行定义变量 cur_dir_entry_pos 来表示当前目录项的地址，每找到一个目录项就将 cur_dir_entry_pos 加上一个目录项大小，直到 cur_dir_entry_pos 的值等于 dir_pos，这才算找到该返回的目录项。

按着这种思路，在第 345 行开始寻找目录项，遍历所有块。所有目录项必然是在文件大小之内，因此在第 346 行对其判断，若如果 dir_pos 大于等于文件尺寸，这说明已经遍历了所有的目录项，直接返回 NULL。提示一下，dir_pos 在执行 sys_opendir 时就被置为 0 了。

接着是将扇区读入到 dir_e 中，遍历所有目录项，第 358 行，只要目录项有效，即目录项的 f_type 不等于 FT_UNKNOWN（FT_UNKNOWN 值为 0），就用当前目录项地址 cur_dir_entry_pos 和 dir->dir_pos 比较，若 cur_dir_entry_pos 小于 dir->dir_pos，这说明都是之前返回过的目录项，因此将 cur_dir_entry_pos 加上目录项大小，并使目录项索引 dir_entry_idx 加 1 后，跳过当前目录项，直到 cur_dir_entry_pos 等于 dir->dir_pos，这才找到了该返回的目录项。

随后在第 366 行把 dir->dir_pos 加上一个目录项大小，使其指向下一个待返回的目录项。最后第 367 行返回目录项地址 dir_e + dir_entry_idx。

好啦，dir_read 到这就介绍完了，大伙儿先休息，一会再续。

14.12.3　实现 sys_readdir 及 sys_rewinddir

在 Linux 中读取目录的函数是 readdir，其原型是：“struct dirent *readdir(DIR *dirp)”，我们将来的 readdir 也是按照此接口来实现的。

遍历目录的操作中，经常会用到目录回绕的功能，也就是使目录的“游标”dir_pos 回到 0，它与 lseek 功能类似，只不过是针对目录的，避免了将目录先关闭再重新打开的繁琐。在 Linux 中目录回绕是用函数 rewinddir 实现的，其原型是：“void rewinddir(DIR *dirp)”，我们也按照此形式实现。

目前还是先实现这两个系统调用的内核部分——sys_readdir 和 sys_rewinddir，代码走起，见代码 14-45。

代码 14-45　（project/c14/k/fs/fs.c）

```
…略
654 /* 读取目录 dir 的 1 个目录项，成功后返回其目录项地址，
       到目录尾时或出错时返回 NULL */
655 struct dir_entry* sys_readdir(struct dir* dir) {
656     ASSERT(dir != NULL);
657     return dir_read(dir);
658 }
659
660 /* 把目录 dir 的指针 dir_pos 置 0 */
661 void sys_rewinddir(struct dir* dir) {
662     dir->dir_pos = 0;
663 }
…略
```

这两个函数都极其简单，甚至有时候我都为代码太少而惭愧。

sys_readdir 是 dir_read 的封装，就这一句实质代码，其功能和 dir_read 是一样的，之所以还要单独创建 sys_readdir，完全是为了系统调用的内核实现部分命名统一。

sys_rewinddir 更简单，它直接将参数 dir 的“游标”dir_pos 置为 0。

下面是功能验证部分，我们在 main.c 中添加调用代码，如代码 14-46 所示。

代码 14-46　（project/c14/k/kernel/main.c）

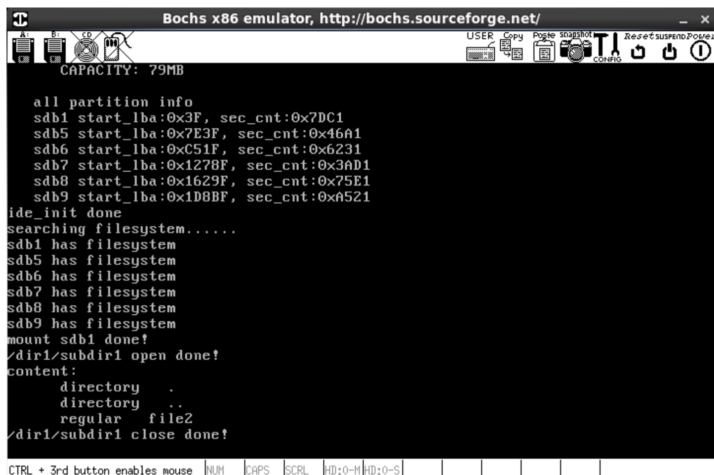
```
…略
19     int main(void) {
20     put_str("I am kernel\n");
21     init_all();
22     /********  测试代码  ********/
23     struct dir* p_dir = sys_opendir("/dir1/subdir1");
24     if (p_dir) {
25         printf("/dir1/subdir1 open done!\ncontent:\n");
26         char* type = NULL;
27         struct dir_entry* dir_e = NULL;
28         while((dir_e = sys_readdir(p_dir))) {
29            if (dir_e->f_type == FT_REGULAR) {
30             type = "regular";
31           } else {
32              type = "directory";
33            }
34            printf("     %s   %s\n", type, dir_e->filename);
35        }
36        if (sys_closedir(p_dir) == 0) {
37            printf("/dir1/subdir1 close done!\n");
38        } else {
39            printf("/dir1/subdir1 close fail!\n");
40        }
41     } else {
42        printf("/dir1/subdir1 open fail!\n");
43     }
44     /********  测试代码  ********/
45     while(1);
46     return 0;
47 }
…略
```

这次我们把两个进程和两个线程的创建代码去掉了，因为在调度的时候会交替输出信息，影响咱们观察调试代码输出的内容。

代码第 23～44 行是打开目录"/dir1/subdir1"，然后输出目录内容：“文件类型文件名”，即如果文件是目录，就输出“directory 文件名”，如果文件是普通文件，就输出“regular 文件名”。这里并没有验证 sys_rewinddir 函数，本节用它的话太牵强，下节更合适。好啦，代码没法再清晰了，咱们直接看输出吧，如图 14-41 所示。

图中输出“/dir1/subdir1”中的 3 个文件，包括两个目录：“.”和“..”，另一个是普通文件 file2，此文件是咱们不久前刚创建的。

输出的信息还是符合预期的，您也猜到了，本节结束啦。



▲图 14-41　遍历目录

14.13 删除目录

我们刚刚创建的目录立即就要在本节中删除了，好可惜。删除目录是目录操作最基本的功能，除此之外，我们还会顺便实现目录回绕。

14.13.1　删除目录与判断空目录

我们有这样的经验，当删除目录时，如果目录中有文件或子目录，无论是 Windows 下，还是 Linux，都会打印类似这样的提示："删除失败，目录非空"。尽管在 Linux 中可以用 rm –r 来删除非空目录，但这是采用递归（recursive）的方式，先删除目录中的文件后再删除目录，Linux 中真正删除目录的命令是 rmdir，它专用于删除空目录。我们也继续这种做法，在删除目录时先判断目录是否为空，不允许删除非空目录。

本节我们在 dir.c 中增加两个函数，见代码 14-47。

代码 14-47　（project/c14/l/fs/dir.c）

```
…略
376 /* 判断目录是否为空 */
377 bool dir_is_empty(struct dir* dir) {
378    struct inode* dir_inode = dir->inode;
379    /* 若目录下只有.和..这两个目录项，则目录为空 */
380    return (dir_inode->i_size == cur_part->sb->dir_entry_size * 2);
381 }
382
383 /* 在父目录 parent_dir 中删除 child_dir */
384 int32_t dir_remove(struct dir* parent_dir, struct dir* child_dir) {
385    struct inode* child_dir_inode  = child_dir->inode;
386    /* 空目录只在 inode->i_sectors[0]中有扇区，其他扇区都应该为空 */
387    int32_t block_idx = 1;
388    while (block_idx < 13) {
389       ASSERT(child_dir_inode->i_sectors[block_idx] == 0);
390       block_idx++;
391    }
392    void* io_buf = sys_malloc(SECTOR_SIZE * 2);
393    if (io_buf == NULL) {
394       printk("dir_remove: malloc for io_buf failed\n");
395       return -1;
396    }
397
398    /* 在父目录 parent_dir 中删除子目录 child_dir 对应的目录项 */
399    delete_dir_entry(cur_part, parent_dir, child_dir_inode->i_no, io_buf);
400
401    /* 回收 inode 中 i_secotrs 中所占用的扇区，
       并同步 inode_bitmap 和 block_bitmap */
```

```
402     inode_release(cur_part, child_dir_inode->i_no);
403     sys_free(io_buf);
404     return 0;
405 }
…略
```

函数 dir_is_empty 接受 1 个参数，目录指针 dir，功能是判断目录 dir 是否为空。其原理很简单，任何目录中都有“.”和“..”这两个目录项，空目录中只剩下这两个目录项，因此若目录的大小等于 2 个目录项的大小，就表示该目录为空。目录为空返回 true，否则返回 false。

函数 dir_remove 接受 2 个参数，父目录指针 parent_dir 和子目录指针 child_dir，功能是在父目录 parent_dir 中删除 child_dir。成功删除则返回 0，否则返回−1。

代码开头通过 ASSERT 判断子目录只有一个块，如注释所说，我们要删除的 child_dir 肯定得是个空目录，空目录只有在其 inode 的 i_sectors[0]中有扇区地址，因此 while 循环中是排查 i_sectors[1～12]。

接下来为 delete_dir_entry 申请缓冲区 io_buf，然后调用 delete_dir_entry 在父目录 parent_dir 中删除子目录 child_dir 对应的目录项，最后调用 inode_release 释放子目录的 inode。

释放 io_buf 后，通过 return 返回 0，函数结束了，本节也悄悄地结束了。

14.13.2 实现 sys_rmdir 及功能验证

目录能创建就得能删除，Linux 下删除目录的函数是 rmdir，其原型是：

int rmdir(const char *pathname)，还是老样子，我们先在 fs.c 中实现其内核部分——sys_rmdir，见代码 14-48。

代码 14-48 （project/c14/l/fs/fs.c）

```
…略
665 /* 删除空目录，成功时返回 0，失败时返回-1*/
666 int32_t sys_rmdir(const char* pathname) {
667     /* 先检查待删除的文件是否存在 */
668     struct path_search_record searched_record;
669     memset(&searched_record, 0, sizeof(struct path_search_record));
670     int inode_no = search_file(pathname, &searched_record);
671     ASSERT(inode_no != 0);
672     int retval = -1;     // 默认返回值
673     if (inode_no == -1) {
674         printk("In %s, sub path %s not exist\n",
             pathname, searched_record.searched_path);
675     } else {
676         if (searched_record.file_type == FT_REGULAR) {
677             printk("%s is regular file!\n", pathname);
678         } else {
679             struct dir* dir = dir_open(cur_part, inode_no);
680             if (!dir_is_empty(dir)) {      // 非空目录不可删除
681                 printk("dir %s is not empty, it is not allowed
                 to delete a nonempty directory!\n", pathname);
682             } else {
683                 if (!dir_remove(searched_record.parent_dir, dir)) {
684                     retval = 0;
685                 }
686             }
687             dir_close(dir);
688         }
689      }
690      dir_close(searched_record.parent_dir);
691      return retval;
692 }
…略
```

大伙儿早已经发现，经过前期大量的基础构建，后面新加的功能代码都不是很长，不禁有些幸福感，咳咳，说代码。

sys_rmdir 接受 1 个参数，待删除的目录 pathname，功能是删除空目录 pathname，成功时返回 0，失败时返回−1。

删除目录之前要确认目录在文件系统上存在，代码第 668～670 行判断目录是否存在。第 672 行定义

了返回值 retval，初始化为−1，以下各步骤若失败将其作为返回值。

在执行 search_file 后，如果返回值 inode_no 为−1，这说明未找到该目录，输出提示信息，执行流跳到 690 行，关闭目录 searched_record.parent_dir。如果 inode_no 不等于−1，这说明找到了 pathname，接下来在第 676～687 行判断 pathname 是目录，还是普通文件。如果 searched_record.file_type 的值是 FT_REGULAR，这说明 pathname 是同名的普通文件，输出提示后跳到第 690 行。否则 pathname 是目录文件，这时候我们将 pathname 打开，然后调用 dir_is_empty 判断其是否为空，如果非空就输出提示，不允许删除非空目录。如果 pathname 为空，就执行 dir_remove 将其在硬盘上删除，如果删除成功，就将 retval 置为 0，流程跳到第 687 行，执行 dir_close 关闭目录在内存中的资源，最后关闭父目录。

下面到了测试时间，main.c 如代码 14-49 所示。

代码 14-49　（project/c14/l/kernel/main.c）

```
…略
19 int main(void) {
20      put_str("I am kernel\n");
21      init_all();
22 /********  测试代码  ********/
23      printf("/dir1 content before delete /dir1/subdir1:\n");
24      struct dir* dir = sys_opendir("/dir1/");
25      char* type = NULL;
26      struct dir_entry* dir_e = NULL;
27      while((dir_e = sys_readdir(dir))) {
28          if (dir_e->f_type == FT_REGULAR) {
29              type = "regular";
30          } else {
31              type = "directory";
32          }
33          printf("     %s   %s\n", type, dir_e->filename);
34     }
35     printf("try to delete nonempty directory /dir1/subdir1\n");
36     if (sys_rmdir("/dir1/subdir1") == -1) {
37         printf("sys_rmdir: /dir1/subdir1 delete fail!\n");
38     }
39
40     printf("try to delete /dir1/subdir1/file2\n");
41     if (sys_rmdir("/dir1/subdir1/file2") == -1) {
42         printf("sys_rmdir: /dir1/subdir1/file2 delete fail!\n");
43     }
44     if (sys_unlink("/dir1/subdir1/file2") == 0 ) {
45         printf("sys_unlink: /dir1/subdir1/file2 delete done\n");
46     }
47
48     printf("try to delete directory /dir1/subdir1 again\n");
49     if (sys_rmdir("/dir1/subdir1") == 0) {
50         printf("/dir1/subdir1 delete done!\n");
51     }
52
53     printf("/dir1 content after delete /dir1/subdir1:\n");
54     sys_rewinddir(dir);
55     while((dir_e = sys_readdir(dir))) {
56        if (dir_e->f_type == FT_REGULAR) {
57            type = "regular";
58        } else {
59            type = "directory";
60        }
61        printf("     %s   %s\n", type, dir_e->filename);
62     }
63
64 /********  测试代码  ********/
65     while(1);
66     return 0;
67 }
…略
```

测试代码虽然有点多，但都比较直白。

目前硬盘上的数据是根目录中存在普通文件 file1 和目录 dir1，在目录 dir1 中存在 subdir1，在 subdir1 中又存在文件 file2。现在的测试思路是。删除/dir1/subdir1 目录。由于是非空目录，直接调用 sys_rmdir 会出错。下面通过 sys_rmdir 和 sys_unlink 分别删除/dir1/subdir1/file2，检测程序健壮性。最后“历尽千辛万苦”终于把目录/dir1/subdir1 删除，然后再次输出目录/dir1 的内容。运行结果如图 14-42 所示。

```
sdb1 has filesystem
sdb5 has filesystem
sdb6 has filesystem
sdb7 has filesystem
sdb8 has filesystem
sdb9 has filesystem
mount sdb1 done!
/dir1 content before delete /dir1/subdir1:
      directory   .
      directory   ..
      directory   subdir1
try to delete nonempty directory /dir1/subdir1
dir /dir1/subdir1 is not empty, it is not allowed to delete a nonempty directory
!
sys_rmdir: /dir1/subdir1 delete fail!
try to delete /dir1/subdir1/file2
/dir1/subdir1/file2 is regular file!
sys_rmdir: /dir1/subdir1/file2 delete fail!
sys_unlink: /dir1/subdir1/file2 delete done
try to delete directory /dir1/subdir1 again
/dir1/subdir1 delete done!
/dir1 content after delete /dir1/subdir1:
      directory   .
      directory   ..
```

▲图 14-42 删除目录演示

大伙儿自行查看输出结果吧，我初步观察是符合预期的。好啦，本节到这啦，手都软了，键盘敲不动了，大伙儿多保重身体，晚安。

14.14 任务的工作目录

在 Linux 中咱们经常会使用命令 pwd 来显示当前工作目录，还要用 cd 命令来改变工作目录，本节咱们要实现类似的功能。

14.14.1 显示当前工作目录的原理及基础代码

我们在 Linux 操作中，经常会 cd 到任何子目录下工作，如果您的 shell 未配置显示全工作路径的话（可以在系统变量$PS1 中配置），您可能经常用命令 pwd 来显示当前工作路径。

话说这是如何实现的呢？您看，在任何目录中都有目录项“..”，它表示父目录。也就是说，有了“..”，无论我们身处任何一级的子目录，都可以“顺藤摸瓜”找到根目录。因此咱们具体的做法是先通过“..”获取当前目录的父目录，在父目录中搜索当前目录的目录项，从目录项中获取当前目录名称，然后再向上找父目录的父目录，再从中获得父目录的名称……沿着目录树层层而上，就能构建出当前目录的绝对路径。

为了辅助这项工作，有一些基础功能要先完成，它们定义在 fs.c 中，见代码 14-50。

代码 14-50 （project/c14/m/fs/fs.c）

```
…略
694 /* 获得父目录的 inode 编号 */
695 static uint32_t get_parent_dir_inode_nr(uint32_t child_inode_nr,
    void* io_buf) {
696     struct inode* child_dir_inode = inode_open(cur_part, child_inode_nr);
697     /* 目录中的目录项".."中包括父目录 inode 编号，".."位于目录的第 0 块 */
698     uint32_t block_lba = child_dir_inode->i_sectors[0];
699     ASSERT(block_lba >= cur_part->sb->data_start_lba);
700     inode_close(child_dir_inode);
701     ide_read(cur_part->my_disk, block_lba, io_buf, 1);
702     struct dir_entry* dir_e = (struct dir_entry*)io_buf;
703     /* 第 0 个目录项是"."，第 1 个目录项是".." */
704     ASSERT(dir_e[1].i_no < 4096 && dir_e[1].f_type == FT_DIRECTORY);
705     return dir_e[1].i_no;      // 返回..即父目录的 inode 编号
```

```
706 }
707
708 /* 在 inode 编号为 p_inode_nr 的目录中查找
     *  inode 编号为 c_inode_nr 的子目录的名字,
709  * 将名字存入缓冲区 path，成功返回 0，失败返-1 */
710 static int get_child_dir_name(uint32_t p_inode_nr, uint32_t c_inode_nr,\
    char* path, void* io_buf) {
711     struct inode* parent_dir_inode = inode_open(cur_part, p_inode_nr);
712     /* 填充 all_blocks，将该目录的所占扇区地址全部写入 all_blocks */
713     uint8_t block_idx = 0;
714     uint32_t all_blocks[140] = {0}, block_cnt = 12;
715     while (block_idx < 12) {
716        all_blocks[block_idx] = parent_dir_inode->i_sectors[block_idx];
717        block_idx++;
718     }
719     if (parent_dir_inode->i_sectors[12]) {
        // 若包含了一级间接块表，将其读入 all_blocks
720        ide_read(cur_part->my_disk,
           parent_dir_inode->i_sectors[12], all_blocks + 12, 1);
721        block_cnt = 140;
722     }
723     inode_close(parent_dir_inode);
724
725     struct dir_entry* dir_e = (struct dir_entry*)io_buf;
726     uint32_t dir_entry_size = cur_part->sb->dir_entry_size;
727     uint32_t dir_entrys_per_sec = (512 / dir_entry_size);
728     block_idx = 0;
729    /* 遍历所有块 */
730     while(block_idx < block_cnt) {
731        if(all_blocks[block_idx]) {    // 如果相应块不为空，则读入相应块
732           ide_read(cur_part->my_disk, all_blocks[block_idx], io_buf, 1);
733           uint8_t dir_e_idx = 0;
734           /* 遍历每个目录项 */
735           while(dir_e_idx < dir_entrys_per_sec) {
736              if ((dir_e + dir_e_idx)->i_no == c_inode_nr) {
737                  strcat(path, "/");
738                  strcat(path, (dir_e + dir_e_idx)->filename);
739                  return 0;
740              }
741              dir_e_idx++;
742           }
743        }
744        block_idx++;
745     }
746     return -1;
747 }
…略
```

函数 get_parent_dir_inode_nr 接受 2 个参数，子目录 inode 编号 child_inode_nr、缓冲区 io_buf，功能是获得父目录的 inode 编号。

此函数是利用子目录中目录项“..”来实现的，在函数开头先通过 inode_open 获得子目录的 inode，用指针 child_dir_inode 保存其地址。目录项“.”和“..”是在执行 sys_mkdir 创建空目录的时候生成的，它们位于目录第 0 个直接块中，即 i_sectors[0]中，因此第 698～701 行，将该块中的数据读入到 io_buf 中。块中第 0 个目录项是“.”，第 1 个目录项是“..”，第 705 行返回第 1 个目录项的 inode 编号，函数结束。

函数 get_child_dir_name 接受 4 个参数，父目录 inode 编号 p_inode_nr、子目录 inode 编号 c_inode_nr、存储路径的缓冲区 path、硬盘读写缓冲区 io_buf，功能是在 inode 编号为 p_inode_nr 的目录中查找 inode 编号为 c_inode_nr 的子目录，将子目录的名字存入缓冲区 path，成功返回 0，失败返−1。

名称是在目录项中存储，故获取名称必然免不了读取目录项所在的块，因此第 711～722 行先打开父目录的 inode，接着很老套地把目录的所有块地址收集到 all_blocks。

第 730～745 行也很老套，遍历所有块，然后在每一个块中遍历所有目录项。在第 736 行，如果发现目录项的 i_no 等于 c_inode_nr，就在第 737 行，用函数 strcat 将路径分隔符'/'追加到 path 中，然后在下一行将目录项的名称追加到'/'之后，最后通过 return 返回 0，函数结束。

简短说下第 737～738 行的 path 操作，函数 get_child_dir_name 每次只获得一层目录的名称，为构建出目录树，它是由主调函数多次调用的，传入的参数 path 用于拼接完整的绝对路径，因此每次调用 get_child_dir_name 时，path 都是非空的，里面的值已经是部分路径了，因此这里用 strcat 去拼接路径。这样说也许有些抽象，到了后面的应用场合就会清楚了。

代码介绍完了，本节也到此结束了，下节再见。

14.14.2　实现 sys_getcwd

Linux 中用 getcwd 函数来获取当前工作路径，其原型是：

```
"char *getcwd(char *buf, size_t size)"
```

buf 是容纳当前目录绝对路径的缓冲区，getcwd 会将当前工作目录的绝对路径写入 buf 中，size 是 buf 的大小。buf 可以由用户提供，也可以由操作系统提供，如果用户不提供 buf，即传给 buf 的参数为 NULL，系统会通过 malloc 单独分配内存给 buf，之后 getcwd 再将 buf 返回，用户进程记得要用 free 释放。本节咱们要实现其内核部分——sys_getcwd。

在进行开发之前还要修改下 pcb，我们要在 pcb 中加个记录当前工作目录的成员——cwd_inode_nr，用它来记录工作目录的 inode 编号，见代码 14-51。

代码 14-51　（project/c14/m/thread/thread.h）

```
…略
77 /* 进程或线程的 pcb，程序控制块 */
78 struct task_struct {
…略
96     uint32_t cwd_inode_nr;     // 进程所在的工作目录的 inode 编号
97     uint32_t stack_magic;
98 };
…略
```

接着还要修改初始化线程函数 init_thread，为 cwd_inode_nr 初始化，见代码 14-52。

代码 14-52　（project/c14/m/thread/thread.c）

```
…略
68 /* 初始化线程基本信息 */
69 void init_thread(struct task_struct* pthread, char* name, int prio) {
…略
98     pthread->cwd_inode_nr = 0;      // 以根目录作为默认工作路径
99     pthread->stack_magic = 0x19870916;    // 自定义的魔数
100 }
…略
```

第 98 行是将 cwd_inode_nr 初始化为 0，也就是任务默认的工作目录是根目录。

一切就绪之后，下面在 fs.c 中着手编写 sys_getcwd 函数，见代码 14-53。

代码 14-53　（project/c14/m/fs/fs.c）

```
…略
749 /* 把当前工作目录绝对路径写入 buf，size 是 buf 的大小。
750  当 buf 为 NULL 时，由操作系统分配存储工作路径的空间并返回地址，
751  失败则返回 NULL */
752 char* sys_getcwd(char* buf, uint32_t size) {
753     /* 确保 buf 不为空，若用户进程提供的 buf 为 NULL，
754     系统调用 getcwd 中要为用户进程通过 malloc 分配内存 */
755     ASSERT(buf != NULL);
756     void* io_buf = sys_malloc(SECTOR_SIZE);
757     if (io_buf == NULL) {
758         return NULL;
759     }
760 
761     struct task_struct* cur_thread = running_thread();
762     int32_t parent_inode_nr = 0;
763     int32_t child_inode_nr = cur_thread->cwd_inode_nr;
764     ASSERT(child_inode_nr >= 0 && child_inode_nr < 4096);
```

```
        // 最大支持 4096 个 inode
765     /* 若当前目录是根目录，直接返回'/' */
766     if (child_inode_nr == 0) {
767        buf[0] = '/';
768        buf[1] = 0;
769        return buf;
770     }
771
772     memset(buf, 0, size);
773     char full_path_reverse[MAX_PATH_LEN] = {0}; // 用来做全路径缓冲区
774
775     /* 从下往上逐层找父目录，直到找到根目录为止。
776      * 当 child_inode_nr 为根目录的 inode 编号(0)时停止，
777      * 即已经查看完根目录中的目录项 */
778     while ((child_inode_nr)) {
779        parent_inode_nr = get_parent_dir_inode_nr(child_inode_nr, io_buf);
780        if (get_child_dir_name(parent_inode_nr, child_inode_nr,\
           full_path_reverse, io_buf) == -1) { // 或未找到名字，失败退出
781           sys_free(io_buf);
782           return NULL;
783        }
784        child_inode_nr = parent_inode_nr;
785     }
786     ASSERT(strlen(full_path_reverse) <= size);
787 /* 至此 full_path_reverse 中的路径是反着的，
788   * 即子目录在前（左），父目录在后（右），
789   * 现将 full_path_reverse 中的路径反置 */
790     char* last_slash;   // 用于记录字符串中最后一个斜杠地址
791     while ((last_slash = strrchr(full_path_reverse, '/'))) {
792        uint16_t len = strlen(buf);
793        strcpy(buf + len, last_slash);
794     /* 在 full_path_reverse 中添加结束字符，
        作为下一次执行 strcpy 中 last_slash 的边界 */
795        *last_slash = 0;
796     }
797     sys_free(io_buf);
798     return buf;
799 }
…略
```

函数 sys_getcwd 接受两个参数，存储绝对路径的缓冲区 buf、缓冲区大小 size，功能是把当前工作目录的绝对路径写入 buf，成功返回 buf 地址，失败返回 NULL。

函数开头用“ASSERT(buf != NULL)”限制了 buf 不为空，我们说过，buf 可以由用户进程提供，也可以由操作系统提供，若用户进程传给 buf 的实参是 NULL，也就是未提供缓冲区，我们会在系统调用 getcwd 中为 buf 通过 malloc 分配内存。接着是为缓冲区 io_buf 申请 1 扇区大小的内存。

第 761～763 行获得当前任务工作目录的 inode 编号，即存储在 pcb 中的 cwd_inode_nr，将其赋值给 child_inode_nr。

第 766～770 行判断，如果 child_inode_nr 是 0，这说明是根目录的 inode 编号，因此把 buf 直接置为“/”后返回。

接着定义了数组 full_path_reverse[MAX_PATH_LEN]，它用于存储工作目录所在的全路径，即绝对路径，不过从名字上看，它是反转的绝对路径，因此它只是临时数据，一会还要将其反转回来。注意，这里只是反转目录顺序，目录名本身不反转。如若原路径为“/ab/c”，在 full_path_reverse 的将是“/c/ab”，并不是“/c/ba”。

第 755～785 行从当前目录向上回溯，逐层找父目录，一直找到根目录为止。第 779 行调用 get_parent_dir_inode_nr 获得父目录的 inode 编号存入 parent_inode_nr，接着调用 get_child_dir_name 把当前工作目录的名字写入 full_path_reverse 中。然后将 child_inode_nr 更新为 parent_inode_nr 开始下一轮循环。

循环过后，在 full_path_reverse 中得到了绝对路径的反转形式，下面将其转换为正常的顺序。代码第 790～798 行通过 while 循环逐层解析目录名，将最终的路径写入 buf 中。

好啦，sys_getcwd 结束啦，下节咱们实现更改工作目录的函数。

14.14.3 实现 sys_chdir 改变工作目录

Linux 中用函数 chdir 来改变当前工作目录，其原型是：

```
"int chdir(const char *path)"
```

您懂的，下面我们先依照此接口实现 sys_chdir，见代码 14-54。

代码 14-54 （project/c14/m/fs/fs.c）

```
…略
801 /* 更改当前工作目录为绝对路径 path，成功则返回 0，失败返回-1 */
802 int32_t sys_chdir(const char* path) {
803     int32_t ret = -1;
804     struct path_search_record searched_record;
805     memset(&searched_record, 0, sizeof(struct path_search_record));
806     int inode_no = search_file(path, &searched_record);
807     if (inode_no != -1) {
808         if (searched_record.file_type == FT_DIRECTORY) {
809             running_thread()->cwd_inode_nr = inode_no;
810             ret = 0;
811         } else {
812             printk("sys_chdir: %s is regular file or other!\n", path);
813         }
814     }
815     dir_close(searched_record.parent_dir);
816     return ret;
817 }
…略
```

函数 sys_chdir 接受 1 个参数，新工作目录的绝对路径 path，功能是更改当前工作目录为绝对路径 path，成功则返回 0，失败返回−1。

任务的工作目录记录在 pcb 中的 cwd_inode_nr，因此更改工作目录的核心原理就是修改 cwd_inode_nr。

工作目录必须是在硬盘上存在的，因此在更改工作目录之前，先要保证新路径 path 是存在的。第 804～806 行搜索 path，如果未找到，也就是返回值 inode_no 为−1，直接返回默认的返回值 ret，即−1。

如果找到了 path，在第 808～813 行要确认 path 是否为目录，万一要是普通文件也会失败。第 808 行判断如果是目录，就用目录 path 的 inode 号 inode_no 给任务的 cwd_inode_nr 赋值，从而完成了工作目录的更改，然后将返回值 ret 置为 0。

最后在第 815 行关闭目录，返回 ret。

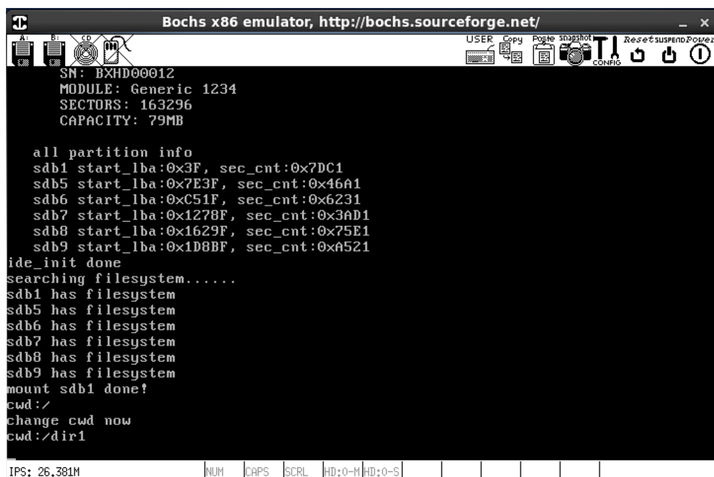
函数介绍完了，下面是测试时间，今天的 main.c 长这样，见代码 14-55。

代码 14-55 （project/c14/m/kernel/main.c）

```
…略
19 int main(void) {
20       put_str("I am kernel\n");
21       init_all();
22 /********  测试代码  ********/
23       char cwd_buf[32] = {0};
24       sys_getcwd(cwd_buf, 32);
25       printf("cwd:%s\n", cwd_buf);
26       sys_chdir("/dir1");
27       printf("change cwd now\n");
28       sys_getcwd(cwd_buf, 32);
29       printf("cwd:%s\n", cwd_buf);
30 /********  测试代码  ********/
31       while(1);
32       return 0;
33 }
…略
```

测试代码中做了三件事，先在第 24～25 行获得当前工作目录并输出，然后第 26 行把工作目录改为/dir1，最后在第 28～29 行再次获得当前工作目录并输出。运行结果如图 14-43 所示。

图中最下面的 3 行是测试结果，目测符合预期，好啦收工啦。



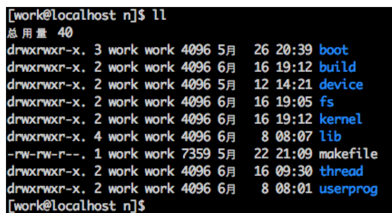
▲图 14-43　获取工作目录与改变工作目录

14.15 获得文件属性

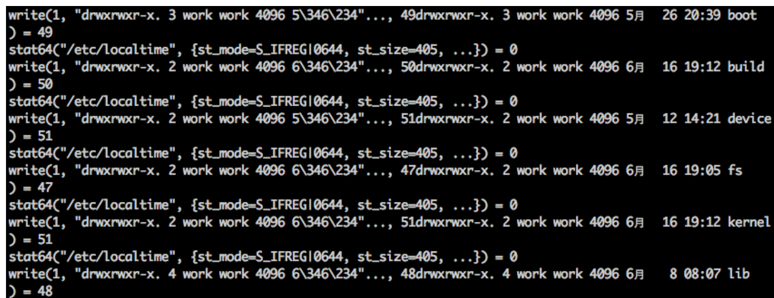
14.15.1　ls 命令的幕后功臣

我们在 shell 中执行 ls 命令时，屏幕会输出文件的属性信息，如图 14-44 所示。

图中的命令 ll 是 ls –l 的别名，执行后输出了文件的类型、权限、属主、时间等。这是如何做到的呢？不妨用 strace 跟一下 ls 命令的执行过程，部分输出如图 14-45 所示。

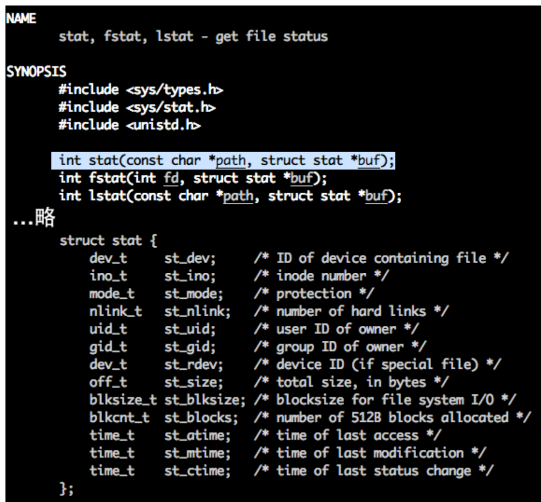


▲图 14-44　文件属性



▲图 14-45　ls 命令中的 stat 系统调用

原来在 ls 命令中调用了大量的系统调用 stat64 和 write，其中 stat64 用于获得文件的属性信息，write 用于把信息输出到屏幕，即标准输出。这里的 stat64 表示 64 位版本的 stat。stat 是干吗的呢？那咱们查看下系统调用 stat 的帮助吧，还是老样子执行 man 2 stat，如图 14-46 所示。



▲图 14-46　stat 帮助信息

图中高亮的是 stat 的原型，参数 path 表示待获取属性的文件，buf 是存储属性的缓冲区，功能是把文件 path 的属性写入 buf，成功后返回 0，失败返回−1。fstat 和 lstat 是 stat 的变体，只是调用形式不同而已，fstat 以文件描述符的形式获取文件属性，因此其参数是 fd，lstat 专用于文件是符号链接的情况。

参数 buf 的类型是 struct stat，下面也贴出了其结构，其中 st_dev 是文件的设备 id，st_ino 是文件的 inode 编号，st_mode 是文件的类型及访问权限的编码，其他不再列举。

说了这些，大伙儿已经了解到，要想在用户环境下获得文件属性，需要提供类 stat 的系统调用，好吧，咱们就直接实现 stat，下节咱们正式编码。

14.15.2 实现 sys_stat

要实现 sys_stat，就要先实现 struct stat 结构，不过咱们的 struct stat 可比 Linux 的简单多了，毕竟咱们不支持访问权限、属主、时间等，好，它定义在 fs.h 中，见代码 14-56。

代码 14-56 （project/c14/n/fs/fs.h）

```
…略
41 /* 文件属性结构体 */
42 struct stat {
43     uint32_t st_ino;              // inode 编号
44     uint32_t st_size;             // 尺寸
45     enum file_types st_filetype;  // 文件类型
46 };
…略
```

咱们的 struct stat 很简单，只有 3 个成员，因此只能获得 3 个属性，有总比没有好^^。成员 st_ino 是文件的 inode 编号，st_size 是文件的字节大小，st_filetype 是文件类型，如普通文件，还是目录。咱们下面看 fs.c 中 sys_stat 的实现，见代码 14-57。

代码 14-57 （project/c14/n/fs/fs.c）

```
…略
819 /*在 buf 中填充文件结构相关信息，成功时返回 0，失败返回-1 */
820 int32_t sys_stat(const char* path, struct stat* buf) {
821     /* 若直接查看根目录'/' */
822     if (!strcmp(path, "/") || !strcmp(path, "/.") || !strcmp(path, "/..")) {
823         buf->st_filetype = FT_DIRECTORY;
824         buf->st_ino = 0;
825         buf->st_size = root_dir.inode->i_size;
826         return 0;
827     }
828
829     int32_t ret = -1;    // 默认返回值
830     struct path_search_record searched_record;
831     memset(&searched_record, 0, sizeof(struct path_search_record));
        // 记得初始化或清 0，否则栈中信息不知道是什么
832     int inode_no = search_file(path, &searched_record);
833     if (inode_no != -1) {
834         struct inode* obj_inode = inode_open(cur_part, inode_no);
        // 只为获得文件大小
835         buf->st_size = obj_inode->i_size;
836         inode_close(obj_inode);
837         buf->st_filetype = searched_record.file_type;
838         buf->st_ino = inode_no;
839         ret = 0;
840     } else {
841         printk("sys_stat: %s not found\n", path);
842     }
843     dir_close(searched_record.parent_dir);
844     return ret;
845 }
…略
```

函数 sys_stat 接受 2 个参数，待获取属性的文件路径 path、存储属性的缓冲区 buf，功能是在 buf 中填充文件结构相关信息，成功时返回 0，失败返回−1。

函数开头判断 path 是否为根目录'/'，如果是就直接在 buf 中写入根目录的信息并成功返回。

第 829 行定义了返回值 ret，默认为−1。第 830～832 行在文件系统上查找文件 path，如果文件存在，第 834 行打开文件的 inode，这是为了获取文件大小。下面分别填充 buf 中的 st_size、st_filetype 和 st_ino，并将 ret 置为 0。如果文件不存在，输出文件不存在的提示。

最后关闭目录，返回 ret，函数结束。

您懂的，每到这个时候就要测试了，main.c 中的测试代码如下。

```
…略
19 int main(void) {
20     put_str("I am kernel\n");
21     init_all();
22 /********  测试代码  ********/
23     struct stat obj_stat;
24     sys_stat("/", &obj_stat);
25     printf("/`s info\n   i_no:%d\n   size:%d\n   filetype:%s\n", \
26            obj_stat.st_ino, obj_stat.st_size, \
27            obj_stat.st_filetype == 2 ? "directory" : "regular");
28     sys_stat("/dir1", &obj_stat);
29     printf("/dir1`s info\n   i_no:%d\n   size:%d\n   filetype:%s\n", \
30            obj_stat.st_ino, obj_stat.st_size, \
31            obj_stat.st_filetype == 2 ? "directory" : "regular");
32 /********  测试代码  ********/
33     while(1);
34     return 0;
35 }
…略
```

第 23 行定义的 obj_stat 将作为 buf 的实参，用于存储文件的属性，接下来调用 sys_stat 分别获取根目录“/”和目录“/dir1”的信息。不说了，看运行结果，如图 14-47 所示。

```
Bochs x86 emulator, http://bochs.sourceforge.net/
   all partition info
   sdb1 start_lba:0x3F, sec_cnt:0x7DC1
   sdb5 start_lba:0x7E3F, sec_cnt:0x46A1
   sdb6 start_lba:0xC51F, sec_cnt:0x6231
   sdb7 start_lba:0x1278F, sec_cnt:0x3AD1
   sdb8 start_lba:0x1629F, sec_cnt:0x75E1
   sdb9 start_lba:0x1D8BF, sec_cnt:0xA521
ide_init done
searching filesystem......
sdb1 has filesystem
sdb5 has filesystem
sdb6 has filesystem
sdb7 has filesystem
sdb8 has filesystem
sdb9 has filesystem
mount sdb1 done!
/`s info
   i_no:0
   size:96
   filetype:directory
/dir1`s info
   i_no:2
   size:48
   filetype:directory
CTRL + 3rd button enables mouse
```

▲图 14-47　sys_stat 运行结果

图中最下面的 8 行是测试代码的输出，根目录的 i_no 为 0，size 为 96，每个目录项大小是 24 字节，因此根目录中共 4 个目录项，分别是“.”、“..”、“file1”、“dir1”。

/dir1 的 i_no 为 2，我们之前在排查根目录时就知道啦（不信的话可以再看下图 14-34），size 是 48，这说明该目录是空目录，其下只有“.”和“..”。

第 15 章　系统交互

15.1 fork 的原理与实现

fork 是什么？叉子？对，fork 就是叉子，没有比这个更完美的答案了。

15.1.1　什么是 fork

开门见山，本节献给那些对 fork 函数不熟悉的同学，高手请略过。

fork 函数原型是 pid_t fork(void)，返回值是数字，该数字有可能是子进程的 pid，有可能是 0，也有可能是−1。1 个函数有 3 种返回值，这是为什么呢？可能的原因是 Linux 中没有获取子进程 pid 的方法，因此，为了让父进程获知自己的孩子是谁，fork 会给父进程返回子进程的 pid。子进程可以通过系统调用 getppid 获知自己的父亲是谁，并且没有 pid 为 0 的进程，因此 fork 给子进程返回 0，以从返回值上和父进程区分开来。如果 fork 失败了，返回的数字便是−1，自然也没有子进程产生，duang…返回值的意义搞定。

虽然解释了返回值的意义，但我还是不理解这其中的奥妙，回想当初我第一次接触 fork 函数的时候，真地觉得它是个高端大气上档次的玩意儿，最让我不可思议的是 fork 执行一次竟然会返回两次 pid，这是什么原理？下面咱们复习下那个经典且让人“迷惑”的例子。

测试代码 fork_a.c

```
 1 #include <unistd.h>
 2 #include <stdio.h>
 3 int main() {
 4     printf("I will fork in 5 seconds\n");
 5     sleep(5);
 6     int pid = fork();
 7     if (pid == -1) {
 8         return 1;
 9     }
10     if (pid) {
11         printf("I am father, my pid is %d\n",getpid());
12         sleep(5);
13         return 0;
14     } else {
15         printf("I am child, my pid is %d\n",getpid());
16         sleep(5);
17         return 0;
18     }
19 }
```

声明下，我是为了说明问题故意把代码写成这样的，我们先不着急看运行结果，简单分析下代码。

主函数开头先输出，5 秒后将执行 fork。然后休眠 5 秒，这样做的目的是预留出缓冲时间来抓取进程的个数，一会您就知道了。

在第 6 行 fork 函数调用过后，后面会根据 pid 的结果跳到不同的执行分支。如果 pid 为−1，这说明 fork 失败，直接退出。到了第 10 行之后，此时的 pid 应该大于等于 0。第 10 行判断如果 pid 为大于 0（非 0），这说明 fork 认为自己是父进程，于是咱们在代码中输出“I am father…”。否则 pid 为 0 的话，这说明 fork 认为自己是子进程，于是输出“I am child…”。下面看运行结果，如图 15-1 所示。

ps 是获取进程状态的命令，我们可以把 ps 的结果通过管道传给 grep 命令，让 grep 过滤出关键信息。

```
[work@localhost book]$ gcc -o fork_test_a fork_a.c
[work@localhost book]$ ./fork_test_a
I will fork in 5 seconds
I am father, my pid is 2810
I am child, my pid is 2815
[work@localhost book]$ _
 在另一窗口中执行
[work@localhost ~]$ ps -ef|grep fork_test|grep -v grep
work      2810  1949  0 16:59 pts/0    00:00:00 ./fork_test_a
[work@localhost ~]$ ps -ef|grep fork_test|grep -v grep
work      2810  1949  0 16:59 pts/0    00:00:00 ./fork_test_a
work      2815  2810  0 16:59 pts/0    00:00:00 ./fork_test_a
[work@localhost ~]$
```

▲图 15-1　fork 测试 a

您看 fork_a.c 编译为 fork_test_a 之后，运行结果中，先打印出了“I will fork in 5 seconds”，然后休眠，与此同时，我在另一个窗口中通过“ps -ef|grep fork_test|grep -v grep”获取了 fork_test 的情况，此时由于尚未执行 fork 函数，系统中就只有一个 fork_test_a。图中画下画线的是同一时间内的配合输出。

过了 5 秒后，父子进程分别打印了自己的 pid，父进程 pid 是 2810，子进程 pid 是 2815，然后同时休眠。此时我又在另一窗口中重复执行以上的命令，屏幕显示了两个同名为 fork_test_a 的进程，也就是下画线下面的输出信息，其中画框框的部分是两个进程各自的 pid。

说到这里，答案似乎清楚一些了，原来 fork 之后，由之前的一个进程变成了两个进程，这说明 fork 的作用就是克隆进程。程序从第 6 行开始便玩起了分身术，从一个进程变成了两个进程，也就是说，内存中多了一个进程，进程拥有独立的地址空间，因此两个进程执行的是独立且相同的代码，也就是两套代码，而且它们各自的指令中都包括第 6 行的 fork 调用，只是子进程是在 fork 函数返回之后才开始执行的，因此执行的是 fork 之后的代码（其实可以让子进程的执行流回到 fork 之前重新调用 fork，但意义不大且非常麻烦），所以在 fork 之后，父子进程像是“分道扬镳”了。

程序是指在磁盘上存储的文件，是静态的，进程是指程序被加载到内存后，在内存中运行中的程序映像，简而言之，进程就是运行的程序。父子进程的进程体（代码段数据段等）是一模一样的，相当于执行了同一程序的两个实例进程。举个例子，比如程序 a 会根据用户输入的不同值而执行不同的分支（代码块），程序 a 加载运行后就成了进程 a，在进程 a 运行的同时又将程序 a 再一次加载到内存运行，这时系统中就又多了一个进程 a，也就是说内存中存在两个同名的进程 a，它们拥有一模一样的程序体，但由于用户输入的内容不同，这两个相同的进程会执行各自程序体中不同的分支，但是这些分支对每个进程而言都是可见的，每个进程中的分支情况都一样，尽管只会选择其中的一个分支来执行。

回到咱们的例子，fork 之后父子进程都要处理各自进程中从第 6 行以后的代码，这些代码对于父子进程来说都是相同的，我们只是把 pid 的值作为分支的条件而已，原则上不需要按照 pid 的不同来划分代码块，这只是设计逻辑的需要，与 fork 无关，fork 的任务就是克隆一个一模一样的进程出来，该进程拥有独立完整的程序体，是个独立的执行流。

按照 pid 来划分程序分支，这只是程序逻辑上的需要，它很经典却又让人“迷惑”，话说不知道当年有多少人和我一样被这个例子搞得蒙圈了。经典是指这充分体现了 fork 的优势，可以同时做两件事。迷惑是指一部分人都会觉得：“fork 之后父子进程一定会按照 pid 分别执行不同的分支，父子进程不能执行同一个代码块”。究其原因，这都是第 10～14 行的 if（pid）…else 惹的祸，关键是大家都用这种例子来介绍 fork，先入为主的影响太深刻了。为了还原其本质，下面把代码搞得更清楚一些。

测试代码 fork_b.c

```
1 #include <unistd.h>
2 #include <stdio.h>
3 int main() {
4     int pid = fork();
5     if (pid == -1) {
```

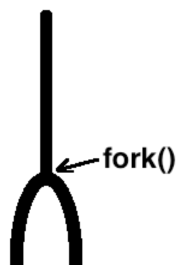
```
 6          return 1;
 7      }
 8      printf("who am I ? my pid is %d\n", getpid());
 9      sleep(5);
10      return 0;
11 }
```

除了第 5 行判断是否失败外，程序中没有任何分支，也就是父子进程都做同样的事，都输出自己的 pid，都 sleep(5)，这样是不是更好理解 fork 了呢？下面是运行结果，如图 15-2 所示。

您看两个进程都打印出“who am I…”并且输出自己的 pid。总之父子进程都在做相同的事。总结下，如图 15-3 所示。

```
[work@localhost book]$ gcc fork_b.c -o fork_test_b
[work@localhost book]$ ./fork_test_b
who am I ? my pid is 2733
who am I ? my pid is 2734
[work@localhost book]$
 在另一窗口中执行
[work@localhost ~]$ ps -ef|grep fork_test|grep -v grep
work      2733  1949  0 16:31 pts/0    00:00:00 ./fork_test_b
work      2734  2733  0 16:31 pts/0    00:00:00 ./fork_test_b
[work@localhost ~]$
```

▲图 15-2　fork 测试 b



▲图 15-3　fork 叉子

fork 就是个叉子，叉子的柄部是一根，在某个地方就一分为二为两个叉子，且每个叉子都是一样的，这同程序在调用 fork 前后的执行流状态是一致的。现在您对 fork 为什么叫 fork 是不是特别的认同了？如果还是不容易理解的话，可以认为：fork 就是相当于同一个程序多次加载执行，因此在内存中产生了多个同名进程。

好啦，有关 fork 的内容就介绍到这，本节到此结束，下节再见。

15.1.2　fork 的实现

fork 利用老进程克隆出一个新进程并使新进程执行，新进程之所以能够执行，本质上是它具备程序体，这其中包括代码和数据等资源。因此 fork 就是把某个进程的全部资源复制了一份，然后让处理器的 cs:eip 寄存器指向新进程的指令部分。故：实现 fork 也要分两步，先复制进程资源，然后再跳过去执行。

考虑一下，进程有哪些资源呢？确定了之后咱们就知道该复制什么了，梳理一下。

（1）进程的 pcb，即 task_struct，这是让任务有“存在感”的身份证。

（2）程序体，即代码段数据段等，这是进程的实体。

（3）用户栈，不用说了，编译器会把局部变量在栈中创建，并且函数调用也离不了栈。

（4）内核栈，进入内核态时，一方面要用它来保存上下文环境，另一方面的作用同用户栈一样。

（5）虚拟地址池，每个进程拥有独立的内存空间，其虚拟地址是用虚拟地址池来管理的。

（6）页表，让进程拥有独立的内存空间。

克隆出来的进程该如何执行呢？这个简单，只要将新进程加入到就绪队列中就可以啦，当然提前要把相关的栈准备好才行。

在真正编写 fork 代码之前，咱们还要增加一些基础设施，首先在 thread.h 的 task_struct 中增加了成员“int16_t parent_pid”，它位于 cwd_inode_nr 之后，表示父进程的 pid，也就是自己的父进程是谁。然后在 thread.c 中的 init_thread 函数中增加一句“pthread->parent_pid = −1;”，使任务的父进程默认为−1，−1 表示没有父进程。另外在 thread.c 中还为 fork 专门增加了个分配 pid 的函数，其声明为“pid_t fork_pid(void)”，其实现就是“return allocate_pid();”，仅仅是个简单的封装，这么做的原因是 allocate_pid 是个静态函数，不能被外部调用，同时又不想破坏其原有类型，所以用 fork_pid 封装它。以上的修改比较简单，故未单独贴代码。

下面还要在 memory.c 中增加个函数，见代码 15-1。

代码 15-1　（project/c15/a/kernel/memory.c）

```
…略
232 /* 安装 1 页大小的 vaddr，专门针对 fork 时虚拟地址位图无需操作的情况 */
233 void* get_a_page_without_opvaddrbitmap(\
    enum pool_flags pf, uint32_t vaddr) {
234    struct pool* mem_pool = pf & PF_KERNEL ? &kernel_pool : &user_pool;
235    lock_acquire(&mem_pool->lock);
236    void* page_phyaddr = palloc(mem_pool);
237    if (page_phyaddr == NULL) {
238        lock_release(&mem_pool->lock);
239        return NULL;
240    }
241    page_table_add((void*)vaddr, page_phyaddr);
242    lock_release(&mem_pool->lock);
243    return (void*)vaddr;
244 }
…略
```

函数 get_a_page_without_opvaddrbitmap 名字好怪异是吧？抱歉，我实在想不出合适的名字，此函数的功能同 get_a_page 类似，只是少了虚拟地址池位图的操作，为了突显这种意义，所以函数名显得怪异。它接受 2 个参数，内存池标识 pf、虚拟地址 vaddr，功能是为 vaddr 分配一物理页，但无需从虚拟地址内存池中设置位图。本函数的实现是 get_a_page 的后半部分，不解释了。等在 fork 中用到此函数时大伙儿就明白为什么这么做了。

下面在 fork.c 中实现了 fork 的内核部分，sys_fork，见代码 15-2-1。

代码 15-2-1　（project/c15/a/userprog/fork.c）

```
…略
10 extern void intr_exit(void);
11
12 /* 将父进程的 pcb 拷贝给子进程 */
13 static int32_t copy_pcb_vaddrbitmap_stack0(\
    struct task_struct* child_thread, struct task_struct* parent_thread) {
14 /* a 复制 pcb 所在的整个页，里面包含进程 pcb 信息及特级 0 极的栈，
    里面包含了返回地址 */
15    memcpy(child_thread, parent_thread, PG_SIZE);
16    /* 下面再单独修改 */
17    child_thread->pid = fork_pid();
18    child_thread->elapsed_ticks = 0;
19    child_thread->status = TASK_READY;
20    child_thread->ticks = child_thread->priority;  //为新进程把时间片充满
21    child_thread->parent_pid = parent_thread->pid;
22    child_thread->general_tag.prev = \
      child_thread->general_tag.next = NULL;
23    child_thread->all_list_tag.prev = child_thread->all_list_tag.next = NULL;
24    block_desc_init(child_thread->u_block_desc);
25    /* b 复制父进程的虚拟地址池的位图 */
26    uint32_t bitmap_pg_cnt = DIV_ROUND_UP(\
      (0xc0000000 - USER_VADDR_START) / PG_SIZE / 8 , PG_SIZE);
27    void* vaddr_btmp = get_kernel_pages(bitmap_pg_cnt);
28    /* 此时 child_thread->userprog_vaddr.vaddr_bitmap.bits
      还是指向父进程虚拟地址的位图地址
29    下面将 child_thread->userprog_vaddr.vaddr_bitmap.bits
      指向自己的位图 vaddr_btmp */
30    memcpy(vaddr_btmp, \
         child_thread->userprog_vaddr.vaddr_bitmap.bits, \
         bitmap_pg_cnt * PG_SIZE);
31    child_thread->userprog_vaddr.vaddr_bitmap.bits = vaddr_btmp;
32   /* 调试用 */
33    ASSERT(strlen(child_thread->name) < 11);
              // pcb.name 的长度是 16，为避免下面 strcat 越界
34    strcat(child_thread->name,"_fork");
35    return 0;
36 }
37
38    /* 复制子进程的进程体（代码和数据）及用户栈 */
39    static void copy_body_stack3(struct task_struct* child_thread,\
      struct task_struct* parent_thread, void* buf_page) {
40    uint8_t* vaddr_btmp =\
```

```
        parent_thread->userprog_vaddr.vaddr_bitmap.bits;
41      uint32_t btmp_bytes_len = \
        parent_thread->userprog_vaddr.vaddr_bitmap.btmp_bytes_len;
42      uint32_t vaddr_start = parent_thread->userprog_vaddr.vaddr_start;
43      uint32_t idx_byte = 0;
44      uint32_t idx_bit = 0;
45      uint32_t prog_vaddr = 0;
46
47      /* 在父进程的用户空间中查找已有数据的页 */
48      while (idx_byte < btmp_bytes_len) {
49        if (vaddr_btmp[idx_byte]) {
50            idx_bit = 0;
51            while (idx_bit < 8) {
52              if ((BITMAP_MASK << idx_bit) & vaddr_btmp[idx_byte]) {
53                  prog_vaddr = (idx_byte * 8 + idx_bit) * PG_SIZE + vaddr_start;
54   /* 下面的操作是将父进程用户空间中的数据通过内核空间做中转，
    最终复制到子进程的用户空间 */
55
56  /* a 将父进程在用户空间中的数据复制到内核缓冲区 buf_page，
57        目的是下面切换到子进程的页表后，还能访问到父进程的数据*/
58                memcpy(buf_page, (void*)prog_vaddr, PG_SIZE);
59
60  /* b 将页表切换到子进程，目的是避免下面申请内存的函数
    将 pte 及 pde 安装在父进程的页表中 */
61                page_dir_activate(child_thread);
62  /* c 申请虚拟地址 prog_vaddr */
63                get_a_page_without_opvaddrbitmap(\
    PF_USER, prog_vaddr);
64
65   /* d 从内核缓冲区中将父进程数据复制到子进程的用户空间 */
66                memcpy((void*)prog_vaddr, buf_page, PG_SIZE);
67
68                /* e 恢复父进程页表 */
69                page_dir_activate(parent_thread);
70            }
71             idx_bit++;
72          }
73        }
74       idx_byte++;
75     }
76 }
…略
```

函数 copy_pcb_vaddrbitmap_stack0 接受 2 个参数，子进程 child_thread、父进程 parent_thread，功能是将父进程的 pcb、虚拟地址位图拷贝给子进程。

函数开头通过 memcpy 把父进程的 pcb 及其内核栈一同复制给子进程。下面开始单独修改 pcb 中的属性值。

第 17 行通过 fork_pid 函数为子进程分配新的 pid，此函数仅是 allocate_pid 的封装，目的是可供外部调用。接下来的第 18～23 行主要是置子进程的 status 为 TASK_READY，目的是让调试器 schedule 安排其上 CPU。还有将子进程时间片 ticks 置为 child_thread->priority，为其加满时间片，以及将 parent_pid 置为 parent_thread->pid 等。

咱们用 child_thread->userprog_vaddr.vaddr_bitmap.bits 来管理进程的虚拟地址空间，此时它还是指向父进程虚拟地址位图所在的内核页框，每个进程都是单独的 4GB 虚拟地址空间，子进程不能和父进程共用同一个虚拟地址位图。因此下面准备复制父进程虚拟地址池的位图给子进程。

第 24 行通过"block_desc_init(child_thread->u_block_desc)"初始化进程自己的内存块描述符，如果没这句代码的话，此处继承的是父进程的块描述符，子进程分配内存时会导致缺页异常。

先在第 26 行计算虚拟地址位图需要的页框数 bitmap_pg_cnt，然后在第 27 行申请 bitmap_pg_cnt 一个内核页框来存储位图。在第 30～31 行完成了虚拟地址位图的复制。

函数内的最后两行是将子进程的函数名加上了后缀"_fork"，按理说子进程与父进程是同名的，因此这是咱们调试用的，以后能在进程列表中看到区别，调试过后会把这两行代码删掉。

下面是函数 copy_body_stack3，它接受 3 个参数，子进程 child_thread、父进程 parent_thread、页缓冲区 buf_page，buf_page 必须是内核页，我们要用它作为所有进程的数据共享缓冲区，后面会有详述。函数

功能是复制子进程的进程体及用户栈。

此函数的主要功能就是拷贝进程的代码和数据资源，也就是复制一份进程体。按理说进程拥有 4GB 的虚拟地址空间，其中低 3GB 的是用户空间，最简单的办法就是将这 3GB 再拷贝一份出来，但是您也笑了，哪有那么大的物理内存，而且咱们还没实现虚拟内存管理机制，即使物理内存很大的话也不能这么败家，整个 32 位地址空间中就一个用户进程……确实这个方法行不通。其实很少有进程把 3GB 用户虚拟地址完全用满的，因此我们只要把用户空间中有效的部分，也就是有数据的部分拷贝出来就行了。

用户使用的内存是用虚拟内存池来管理的，也就是 pcb 中的 userprog_vaddr。这包括用户进程体占用的内存、堆中申请的内存和和用户栈内存。我们之前已经了解过进程的内存布局，其中低 3GB 的虚拟地址空间中，低地址处是进程的数据段、代码段，其余部分是堆和栈共同的空间，堆从低地址往高地址发展，栈从 USER_STACK3_VADDR，即 0xc0000000 - 0x1000 处往低地址发展。它们的分布不连续，因此我们要遍历虚拟地址位图中的每一位，这样才能找出进程正在使用的内存。

第 40～45 行是为后续遍历做准备，将各种变量指向父进程虚拟内存池相关的参数或值。下面开始寻找父进程占用的内存。

我们的目的是将父进程用户空间中的数据复制到子进程的用户空间。但大伙儿知道，各用户进程的低 3GB 空间是独立的，各用户进程不能互相访问彼此的空间，但高 1GB 是内核空间，内核空间是所有用户进程共享的，因此要想把数据从一个进程拷贝到另一个进程，必须要借助内核空间作为数据中转，即先将父进程用户空间中的数据复制到内核的 buf_page 中，然后再将 buf_page 复制到子进程的用户空间中。

为节省缓冲区空间，这里我们采用的方法是：在父进程虚拟地址空间中每找到一页占用的内存，就在子进程的虚拟地址空间中分配一页内存，然后将 buf_page 中父进程的数据复制到为子进程新分配的虚拟地址空间页，也就是一页一页的对拷，因此我们的 buf_page 只要 1 页大小就够了。但是大伙儿一定也猜到了，不同进程之所有拥有单独的虚拟地址空间，原因是它们各自有单独的页目录表，我们在分配内存的时候，会在页表中产生新的 pte，如果申请的内存跨 4MB 的页表大小的话，还要在页目录表中创建 pde，既然我们是为子进程分配内存，那么我们要确保这些 pte 和 pde 是创建在子进程的页目录表中。所以在将 buf_page 的数据拷贝到子进程之前，一定要将页表替换为子进程的页表。思路就是这样，下面继续看代码。

第 48 行在父进程虚拟地址位图字节长度 btmp_bytes_len 的范围内逐字节查看位图，第 49 行如果该字节不为 0，也就是某位为 1，即某个位有效，已分配，下面第 51 行开始逐位查看该字节。通过第 52 行的 if 判断，如果某位的值为 1，就在第 53 行将该位转换为虚拟地址 prog_vaddr，接下来通过 memcpy 将 prog_vaddr 处的 1 页复制到 buf_page，注意此时只是完成了父进程的数据拷贝到内核空间。下面在为子进程分配内存之前，先调用“page_dir_activate(child_thread)”激活子进程的页表，然后再调用“get_a_page_without_opvaddrbitmap(PF_USER, prog_vaddr)”为子进程分配 1 页，接着再调用“memcpy((void*)prog_vaddr, buf_page, PG_SIZE);”完成内核空间到子进程空间的复制，最后再调用“page_dir_activate(parent_thread)”将父进程的页表恢复。然后进入下一循环，继续寻找父进程占用的虚拟空间。

由于用户栈位于低 3GB 虚拟空间中的最高处，所以循环到最后时会完成用户栈的复制。

下面看 fork.c 的第二部分，见代码 15-2-2。

代码 15-2-2　（project/c15/a/userprog/fork.c）

```
…略
 78   /* 为子进程构建 thread_stack 和修改返回值 */
 79   static int32_t build_child_stack(struct task_struct* child_thread) {
 80   /* a 使子进程 pid 返回值为 0 */
 81   /* 获取子进程 0 级栈栈顶 */
 82   struct intr_stack* intr_0_stack = (struct intr_stack*)\
      ((uint32_t)child_thread + PG_SIZE - sizeof(struct intr_stack));
 83     /* 修改子进程的返回值为 0 */
 84     intr_0_stack->eax = 0;
 85
 86     /* b 为 switch_to 构建 struct thread_stack,
        将其构建在紧临 intr_stack 之下的空间*/
 87     uint32_t* ret_addr_in_thread_stack  = (uint32_t*)intr_0_stack - 1;
 88
```

```
 89     /***   这三行不是必要的，只是为了梳理 thread_stack 中的关系 ***/
 90     uint32_t* esi_ptr_in_thread_stack = (uint32_t*)intr_0_stack - 2;
 91     uint32_t* edi_ptr_in_thread_stack = (uint32_t*)intr_0_stack - 3;
 92     uint32_t* ebx_ptr_in_thread_stack = (uint32_t*)intr_0_stack - 4;
 93     /**********************************************************/
 94
 95     /* ebp 在 thread_stack 中的地址便是当时的 esp（0 级栈的栈顶），
 96     即 esp 为"(uint32_t*)intr_0_stack - 5" */
 97     uint32_t* ebp_ptr_in_thread_stack = (uint32_t*)intr_0_stack - 5;
 98
 99     /* switch_to 的返回地址更新为 intr_exit，直接从中断返回 */
100    *ret_addr_in_thread_stack = (uint32_t)intr_exit;
101
102     /* 下面这两行赋值只是为了使构建的 thread_stack 更加清晰，
       其实也不需要，因为在进入 intr_exit 后一系列的 pop
103        会把寄存器中的数据覆盖 */
104     *ebp_ptr_in_thread_stack = *ebx_ptr_in_thread_stack =\
105     *edi_ptr_in_thread_stack = *esi_ptr_in_thread_stack = 0;
106     /*********************************************************/
107
108     /* 把构建的 thread_stack 的栈顶作为 switch_to 恢复数据时的栈顶 */
109     child_thread->self_kstack = ebp_ptr_in_thread_stack;
110     return 0;
111 }
112
113 /* 更新 inode 打开数 */
114 static void update_inode_open_cnts(struct task_struct* thread) {
115      int32_t local_fd = 3, global_fd = 0;
116      while (local_fd < MAX_FILES_OPEN_PER_PROC) {
117         global_fd = thread->fd_table[local_fd];
118         ASSERT(global_fd < MAX_FILE_OPEN);
119         if (global_fd != -1) {
120              file_table[global_fd].fd_inode->i_open_cnts++;
121         }
122         local_fd++;
123      }
124 }
125
126      /* 拷贝父进程本身所占资源给子进程 */
127      static int32_t copy_process(struct task_struct* child_thread,\
         struct task_struct* parent_thread) {
128      /* 内核缓冲区，
        作为父进程用户空间的数据复制到子进程用户空间的中转 */
129     void* buf_page = get_kernel_pages(1);
130     if (buf_page == NULL) {
131         return -1;
132     }
133
134     /* a 复制父进程的 pcb、虚拟地址位图、内核栈到子进程 */
135     if (copy_pcb_vaddrbitmap_stack0(child_thread, parent_thread) == -1) {
136         return -1;
137     }
138
139     /* b 为子进程创建页表，此页表仅包括内核空间 */
140     child_thread->pgdir = create_page_dir();
141     if(child_thread->pgdir == NULL) {
142         return -1;
143     }
144
145     /* c 复制父进程进程体及用户栈给子进程 */
146     copy_body_stack3(child_thread, parent_thread, buf_page);
147
148    /* d 构建子进程 thread_stack 和修改返回值 pid */
149    build_child_stack(child_thread);
150
151    /* e 更新文件 inode 的打开数 */
152    update_inode_open_cnts(child_thread);
153
154    mfree_page(PF_KERNEL, buf_page, 1);
155    return 0;
156 }
157
```

```
158   /* fork 子进程，内核线程不可直接调用 */
159   pid_t sys_fork(void) {
160     struct task_struct* parent_thread = running_thread();
161     struct task_struct* child_thread = get_kernel_pages(1);
      // 为子进程创建 pcb（task_struct 结构）
162     if (child_thread == NULL) {
163         return -1;
164     }
165     ASSERT(INTR_OFF == intr_get_status() &&\
       parent_thread->pgdir != NULL);
166
167     if (copy_process(child_thread, parent_thread) == -1) {
168         return -1;
169     }
170
171     /* 添加到就绪线程队列和所有线程队列，子进程由调试器安排运行 */
172     ASSERT(!elem_find(&thread_ready_list, &child_thread->general_tag));
173     list_append(&thread_ready_list, &child_thread->general_tag);
174     ASSERT(!elem_find(&thread_all_list, &child_thread->all_list_tag));
175     list_append(&thread_all_list, &child_thread->all_list_tag);
176
177     return child_thread->pid;    // 父进程返回子进程的 pid
178 }
```

函数 build_child_stack 接受 1 个参数，子进程 child_thread。功能是为子进程构建 thread_stack 和修改返回值。

大伙儿知道，父进程在执行 fork 系统调用时会进入内核态，中断入口程序会保存父进程的上下文，这其中包括进程在用户态下的 CS:EIP 的值，因此父进程从 fork 系统调用返回后，可以继续 fork 之后的代码执行。问题来了，我们通过例子已经知道，子进程也是从 fork 后的代码处继续运行的，这是怎样做到的呢？

在这之前我们已经通过函数 copy_pcb_vaddrbitmap_stack0 将父进程的内核栈复制到了子进程的内核栈中，那里保存了返回地址，也就是 fork 之后的地址，为了让子进程也能继续 fork 之后的代码运行，咱们必须让它同父进程一样，从中断退出，也就是要经过 intr_exit。

子进程是由调试器 schedule 调度执行的，它要用到 switch_to 函数，而 switch_to 函数要从栈 thread_stack 中恢复上下文，因此我们要想办法构建出合适的 thread_stack。您看，本函数还是有点意思的。

大伙儿还记得 intr_stack 栈是什么吧？就是在 kernel.S 中，中断入口程序 intr%1entry 中保存任务上下文的地方。函数开头先获得子进程的 intr_stack 栈的地址，目的是在下一行将 eax 的值置为 0，原因是 fork 会为子进程返回 0 值，根据 abi 约定，eax 寄存器中是函数返回值，因此第 84 行将 intr_stack 栈中的 eax 置为 0。

下面我们要为 switch_to 函数构建一个 thread_stack，这里咱们把它的栈底放在 intr_stack 栈顶的下面，也就是 intr_0_stack 的地址减 4 字节的地方，即第 87 行的代码“(uint32_t*)intr_0_stack - 1”，此地址是 thread_stack 栈中 eip 的位置。接下来的第 89～97 行分别为 thread_stack 中的 esi、edi、ebx、ebp 安排位置，这里最重要的是第 97 行的指针 ebp_ptr_in_thread_stack，它是 thread_stack 的栈顶，我们必须把它的值存放在 pcb 中偏移为 0 的地方，即 task_struct 中的 self_kstack 处，将来 switch_to 要用它作为栈顶，并且执行一系列的 pop 来恢复上下文。第 90～92 行的 3 个寄存器指针只是为了使 thread_stack 栈显得更加具有可读性，实际运行中不需要它们的具体值。

第 100 行将地址 ret_addr_in_thread_stack 处的值赋值为 intr_exit 的地址，也就是 thread_stack 中的 eip 是 intr_exit，这就保证了子进程被调度时，可以直接从中断返回，也就是实现了从 fork 之后的代码处继续执行的目的。

最后在第 109 行把 ebp_ptr_in_thread_stack 的值，也就是 thread_stack 的栈顶记录在 pcb 的 self_ kstack 处，这样 switch_to 便获得了咱们刚刚构建的 thread_stack 栈顶，从而使程序迈向 intr_exit。

接下来是函数 update_inode_open_cnts，它接受 1 个参数，线程 thread，功能是 fork 之后，更新线程 thread 的 inode 打开数。函数原理也很简单，遍历 fd_table 中除前 3 个标准文件描述符之外的所有文件描述符，从中获得全局文件表 file_table 的下标 global_fd，通过它在 file_table 中找到对应的文件结构，使相应文件结构中 fd_inode 的 i_open_cnts 加 1。

下面看 copy_process 函数，此函数接受 2 个参数，子进程 child_thread 和父进程 parent_thread，功能是拷贝父进程本身所占资源给子进程。

此函数是前面所介绍的函数的封装，函数开头申请了 1 页的内核空间作为内核缓冲区，即 buf_page。然后调用函数 copy_pcb_vaddrbitmap_stack0 把父进程子的 pcb、虚拟地址位图及内核栈复制给子进程，接着调用 create_page_dir 函数为子进程创建页表，该函数定义在 process.c 中，很久以前介绍过。然后调用函数 copy_body_stack3 复制父进程进程体及用户栈给子进程，接着调用函数 build_child_stack 为子进程构建 thread_stack，随后调用 update_inode_open_cnts 更新 inode 的打开数，最后释放 buf_page。

下面是函数 sys_fork，即 fork 的内核实现部分，fork 本身是无参数的，因此 sys_fork 也无参数。功能是克隆当前进程，即父进程。

函数先调用 get_kernel_pages(1) 获得 1 页内核空间作为子进程的 pcb。接下来调用 copy_process 复制父进程的信息到子进程，然后在第 172～175 行将其加入到就绪队列和全部队列，最后返回子进程的 pid。

好啦，fork.c 就介绍完了，本节就到这，咱们下节再会。

15.1.3 添加 fork 系统调用与实现 init 进程

本节咱们要实现 init 进程。在 Linux 中，init 是用户级进程，它是第一个启动的程序，因此它的 pid 是 1，后续的所有进程都是它的孩子，故 init 是所有进程的父进程，所以它还负责所有子进程的资源回收，这一点在以后介绍 wait 时会给大伙详述。

既然 init 是所有进程的父进程，也就是说它要主动调用 fork 才能派生出子子孙孙，所以在实现它之前，咱们要先完成 fork 系统调用。

系统调用的 3 个步骤，顺便说下具体的代码。

（1）在 syscall.h 中的 enum SYSCALL_NR 结构中添加 SYS_FORK。

（2）在 syscall.c 中添加 fork()，原型是 pid_t fork(void)，实现是“return _syscall0(SYS_FORK);”。

（3）在 syscall-init.c 中的函数 syscall_init 中，添加代码“syscall_table[SYS_FORK] = sys_fork;”。

下面咱们看 init 的实现，init 定义在 main.c 中，如代码 15-3 所示。

代码 15-3 （project/c15/a/kernel/main.c）

```
…略
25 /* init 进程 */
26 void init(void) {
27     uint32_t ret_pid = fork();
28     if(ret_pid) {
29         printf("i am father, my pid is %d,
           child pid is %d\n", getpid(), ret_pid);
30     } else {
31         printf("i am child, my pid is %d, ret pid is %d\n", getpid(), ret_pid);
32     }
33     while(1);
34 }
…略
```

init 实现比较简单，它先调用 fork，派生出子进程，然后在父子进程中分别打印自己的 pid 以及 fork 的返回值 ret_pid。

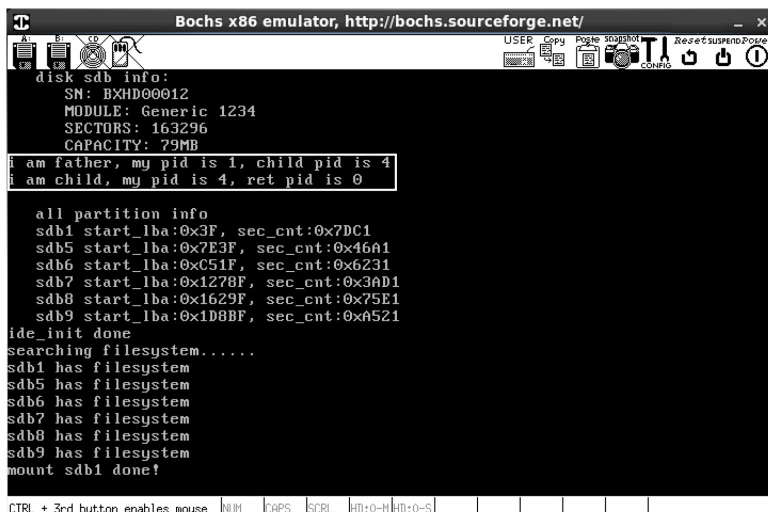
有个问题，init 是用户级进程，因此咱们要调用 process_execute 创建进程，但由谁来创建 init 进程呢？大伙儿知道，pid 是从 1 开始分配的，init 的 pid 是 1，因此咱们得早早地创建 init 进程，抢夺 1 号 pid。目前系统中有主线程，其 pid 为 1，还有 ilde 线程，其 pid 为 2，因此咱们应该在创建主线程的函数 make_main_thread 之前创建 init，也就是在函数 thread_init 中完成，见代码 15-4。

代码 15-4 （project/c15/a/thread/thread.c）

```
…略
214 /* 初始化线程环境 */
215 void thread_init(void) {
…略
```

```
222 /* 先创建第一个用户进程:init */
223   process_execute(init, "init");
     // 放在第一个初始化，这是第一个进程，init 进程的 pid 为 1
224
225 /* 将当前 main 函数创建为线程 */
226   make_main_thread();
227
228 /* 创建 idle 线程 */
229   idle_thread = thread_start("idle", 10, idle, NULL);
230
231   put_str("thread_init done\n");
232 }
…略
```

所有的相关代码都介绍过了，编译运行吧，运行结果如图 15-4 所示。



▲图 15-4　fork 运行结果

init 进程执行的较早，因此它输出的信息混在了硬盘信息中，方框中标出的便是 init 的信息。方框中第一行是父进程，也就是 init 的输出，它的 pid 是 1，子进程 pid 是 4，也就是 fork 给它的返回值是 4。第二行是子进程的输出，它的 pid 是 4，fork 给它的返回值是 0，好啦，目测运行结果符合预期。

也许有读者会认为应该把 init 的进程放在 main.c 中的 init_all() 之后，这样就能避免输出信息混杂在一起了，但这样的话，为保证 init 进程的 pid 为 1，要在 pid 的分配上做些特殊处理，这多少都有些不完美，因此就这样吧，再说了，人家 Linux 的 init 之所以 pid 为 1，原因是 init 真的是第 1 个任务，需要在 init 中完成很多初始化的工作，咱们不要求太高，达到了这个目的就行了。

15.2 添加 read 系统调用，获取键盘输入

让系统同用户交互，首先得知道用户键入了什么命令，然后再对输入进行分析，进而采取相应的处理。因此您肯定想到了，首先要解决的问题是怎样从键盘中获取用户的输入。

Linux 中从键盘获取输入是利用 read 系统调用，咱们在很久之前实现了 sys_read，也许有同学会说，现在只要按照那三个步骤添加 read 系统调用就行了。其实……旧版本的 sys_read 只能从文件中获取数据，还不能从标准输入设备键盘中读取数据，因此当务之急，先要改进 sys_read，让其支持键盘，见代码 15-5。

代码 15-5　（project/c15/b/fs/fs.c）

```
…略
408 /* 从文件描述符 fd 指向的文件中读取 count 个字节到 buf，
    若成功则返回读出的字节数，到文件尾则返回-1 */
```

```
409 int32_t sys_read(int32_t fd, void* buf, uint32_t count) {
410     ASSERT(buf != NULL);
411     int32_t ret = -1;
412     if (fd < 0 || fd == stdout_no || fd == stderr_no) {
413         printk("sys_read: fd error\n");
414     } else if (fd == stdin_no) {
415         char* buffer = buf;
416         uint32_t bytes_read = 0;
417         while (bytes_read < count) {
418            *buffer = ioq_getchar(&kbd_buf);
419            bytes_read++;
420            buffer++;
421         }
422         ret = (bytes_read == 0 ? -1 : (int32_t)bytes_read);
423     } else {
424         uint32_t _fd = fd_local2global(fd);
425         ret = file_read(&file_table[_fd], buf, count);
426     }
427     return ret;
428 }
…略
```

第 414～422 行加入了标准输入 stdin_no 的处理，在第 414 行，若发现 fd 是 stdin_no，下面就通过 while 和 ioq_getchar(&kbd_buf)，每次从键盘缓冲区 kbd_buf 中获取 1 个字符，直到获取了 count 个字符为止。大伙儿还记得吗？kbd_buf 是我们存储键盘输入的环形缓冲区，它定义在 keyboard.c 中，我们已在 keyboard.h 中为其添加了外部声明“extern struct ioqueue kbd_buf;”。

下面我们在 syscall.c 中添加 read 的系统调用，Linux 中 read 函数的原型是：

```
ssize_t read(int fd, void *buf, size_t count);
```

这和 sys_read 接口是一样的（其实咱们所有系统调用的内核实现部分都是模仿 Linux 系统调用接口实现的），我悄悄在 syscall.h 的 enum SYSCALL_NR 中添加 SYS_READ 后（代码就一句，不再单独贴出），下面在 syscall.c 中添加系统调用 read 的实现，见代码 15-6。

代码 15-6 （project/c15/b/lib/user/syscall.c）

```
…略
77 /* 从文件描述符 fd 中读取 count 个字节到 buf */
78 int32_t read(int32_t fd, void* buf, uint32_t count) {
79     return _syscall3(SYS_READ, fd, buf, count);
80 }
```

最后在 syscall_init.c 的 syscall_init 函数中添加代码“syscall_table[SYS_READ] = sys_read”，在 syscall_table 数组中把 read 与 sys_read 绑定到一起就行了。

本节先到这，等以后实现了其他功能咱们一块测试 read 系统调用，好，下节见。

15.3 添加 putchar、clear 系统调用

有时候我们需要输出单个字符，这在 Linux 中是用系统调用 putchar 实现的。它的原型是“int putchar(int c)”，若成功输出，则返回值为 (unsigned int)c，若失败则返回 EOF，即 End Of File，EOF 通常为−1。坦白说，咱们的系统并未考虑到失败的情况，因此不会返回−1。干脆咱们就不按照标准的 putchar 来实现了，咱们的 putchar 无返回值，只会输出一个字符。其实 putchar 对应的内核实现咱们已经有了，可以直接用 console_put_char。

在 Linux 中，当屏幕输出很乱时，咱们通常会执行 clear 命令，或者按快捷键“ctrl+l”来实现清屏。这在幕后是通过 clear 系统调用来实现的，本节咱们也来实现这个功能。

要实现 clear，肯定要先实现其底层，也就是对应的内核实现，这部分还没有现成可用的函数，肯定要新写一个啦。不过与之前不同的是 clear 对应的内核部分并不叫 sys_clear，而是叫 cls_screen，这样显得意义更明确专一（当然您依然可以定义为 sys_clear）。另外稍微有点小恐怖的是它是用汇编代码完成的，它定义在 print.S 中。话说好久不“汇编”了，居然有点心虚的感觉，不过大伙儿不要怕，里面都是曾经介绍过的代

码，而且只是很简单的几行，如代码 15-7 所示，不再解释了。

代码 15-7　（project/c15/b/lib/kernel/print.S）

```
…略
155 global cls_screen
156 cls_screen:
157     pushad
158     ;;;;;;;;;;;;;;;
159       ; 由于用户程序的 cpl 为 3，显存段的 dpl 为 0，
          ;故用于显存段的选择子 gs 在低于自己特权的环境中为 0，
160       ; 导致用户程序再次进入中断后，gs 为 0，
          ; 故直接在 put_str 中每次都为 gs 赋值
161       mov ax, SELECTOR_VIDEO  ; 不能直接把立即数送入 gs，需由 ax 中转
162       mov gs, ax
163
164       mov ebx, 0
165       mov ecx, 80*25
166   .cls:
167       mov word [gs:ebx], 0x0720    ;0x0720 是黑底白字的空格键
168       add ebx, 2
169       loop .cls
170       mov ebx, 0
171
172   .set_cursor:              ;直接把 set_cursor 搬过来用，省事
173 ;;;;;;; 1 先设置高 8 位 ;;;;;;;;
174      mov dx, 0x03d4     ;索引寄存器
175      mov al, 0x0e        ;用于提供光标位置的高 8 位
176      out dx, al
177      mov dx, 0x03d5    ;通过读写数据端口 0x3d5 来获得或设置光标位置
178      mov al, bh
179      out dx, al
180
181 ;;;;;;; 2 再设置低 8 位 ;;;;;;;;;
182      mov dx, 0x03d4
183      mov al, 0x0f
184      out dx, al
185      mov dx, 0x03d5
186      mov al, bl
187      out dx, al
188      popad
189      ret
…略
```

下面是系统调用 putchar 和 clear 的实现，见代码 15-8。

代码 15-8　（project/c15/b/lib/user/syscall.c）

```
…略
82 /* 输出一个字符 */
83 void putchar(char char_asci) {
84     _syscall1(SYS_PUTCHAR, char_asci);
85 }
86
87 /* 清空屏幕 */
88 void clear(void) {
89      _syscall0(SYS_CLEAR);
90 }
…略
```

这两个函数完成之后，还要在 syscall.h 的 enum SYSCALL_NR 结构中添加 SYS_PUTCHAR 和 SYS_CLEAR，最后在 syscall-init.c 中增加初始化代码“syscall_table[SYS_PUTCHAR] = sys_putchar;”和“syscall_table[SYS_CLEAR]　= cls_screen;”。

好啦，基础的部分实现得差不多了，下节咱们开始实现一个简单的 shell，是不是感觉有点突然，哈哈，该来的总会来。

15.4 实现一个简单的 shell

本节咱们实现一个简单的 shell，能处理键入的命令，实现内部命令及外部命令，支持简单的快捷键等。

15.4.1　shell 雏形

操作系统是为用户服务的，要想实现和用户的交互，操作系统得有办法感知用户的输入并给予回馈，也就是必须要为用户提供个交互接口。在 Windows 中，图形界面的资源管理器和命令行窗口都是交互接口，尽管这些交互接口名字及外观各异，但它们往往被统称为“外壳”程序。

想想看动物外壳是什么。外壳也称为甲壳，它的作用一是保护动物内部的器官组织，避免来自外界的破坏，二是通过它来感知外面的世界并给予回应。您看，这其实是典型的操作系统交互接口的作用，因此 Linux 系统采用了更直接的叫法，它的交互接口就叫 shell，直译过来就是“外壳”。shell 的功能大致是获取用户的键入，然后分析输入的字符串，判断是内部命令，还是外部命令，然后执行不同的策略。Linux 中的 shell 有各种版本，大家最常用的应该是 bash，即 Bourne Again shell。好啦，这些常识大伙儿肯定知道，我就不班门弄斧了。

实际的 shell 实现是非常复杂的，咱们的 shell 肯定没那么强大，实现的比较简单，所以大伙儿请放心，咱们一定会过得轻松愉快。本节新建目录 shell，在其下创建文件 shell.c，好啦，上代码，见代码 15-9。

代码 15-9　（project/c15/b/shell/shell.c）

```
…略
11 #define cmd_len 128       // 最大支持键入 128 个字符的命令行输入
12 #define MAX_ARG_NR 16    // 加上命令名外，最多支持 15 个参数
13
14 /* 存储输入的命令 */
15 static char cmd_line[cmd_len] = {0};
16
17    /* 用来记录当前目录，是当前目录的缓存，
   每次执行 cd 命令时会更新此内容 */
18 char cwd_cache[64] = {0};
19
20 /* 输出提示符 */
21 void print_prompt(void) {
22     printf("[rabbit@localhost %s]$ ", cwd_cache);
23 }
24
25 /* 从键盘缓冲区中最多读入 count 个字节到 buf */
26 static void readline(char* buf, int32_t count) {
27     assert(buf != NULL && count > 0);
28     char* pos = buf;
29     while (read(stdin_no, pos, 1) != -1 && (pos - buf) < count) {
                         // 在不出错情况下，直到找到回车符才返回
30        switch (*pos) {
31         /* 找到回车或换行符后认为键入的命令结束，直接返回 */
32           case '\n':
33           case '\r':
34               *pos = 0;      // 添加 cmd_line 的终止字符 0
35               putchar('\n');
36               return;
37
38           case '\b':
39               if (buf[0] != '\b') {// 阻止删除非本次输入的信息
40                    --pos;      // 退回到缓冲区 cmd_line 中上一个字符
41                    putchar('\b');
42               }
43              break;
44
45            /* 非控制键则输出字符 */
46            default:
47                putchar(*pos);
48                pos++;
49        }
50    }
```

```
51     printf("readline: can`t find enter_key in the cmd_line, 
       max num of char is 128\n");
52 }
53
54 /* 简单的 shell */
55 void my_shell(void) {
56     cwd_cache[0] = '/';
57     while (1) {
58        print_prompt();
59        memset(cmd_line, 0, cmd_len);
60        readline(cmd_line, cmd_len);
61        if (cmd_line[0] == 0) {    // 若只键入了一个回车
62            continue;
63        }
64     }
65     panic("my_shell: should not be here");
66 }
```

用户键入命令的命令是以字符串形式提交的，因此其长度是有限制的，下面通过两个宏限制命令串的长度。

第 11 行的 cmd_len 表示命令字符串最大的长度，其值为 128，也就是说用户输入的命令字符串长度是有限的，不能超过 128 个字符。下一行的 MAX_ARG_NR 表示最大支持的参数个数，这里咱们定义为 16。

数组 cmd_line 用来存储键入的命令。数组 cwd_cache 用来存储当前目录名，主要是在命令提示符中，它由以后实现的 cd 命令来维护。

函数 print_prompt 用于输出命令提示符，也就是咱们登录 shell 后，命令行中显示的主机名等，这里咱们用 printf 函数输出"[rabbit@localhost %s]$ "，对此格式大伙儿应该比较熟悉啦，其中"@"左边的是用户名，右边的是主机名，"$"表示普通用户。当然我们没实现用户管理，所以并不会把"$"换成"#"，您懂的，"#"表示 root 账号，它太可怕了，平时可不敢随便用，让我们继续保持这种好习惯，平时就待在低特权级的普通账号下。

函数 readline 接受 2 个参数，缓冲区 buf 和读入的字符数，功能是从键盘缓冲区中最多读入 count 个字节到 buf。

字符指针 pos 指向缓冲区 buf，我们后面将通过 pos 往 buf 中写数据。

函数体中主要是个 while 循环，每次通过 read 系统调用读入 1 个字符到 pos 中，也就是 buf 中。也许您在想，为什么不是一次读入 cmd_len 个字符？这样效率会很高啊。其实，咱们这样做是避免用户输入不足 cmd_len 个字符，而无法得到及时的处理，如果非要等读到回车符才处理，那么用户键入的内容并不是立即反映到屏幕上。您想，按键时屏幕上无显示，用户会感到很迷茫，心里一定在想，你这是几个意思。

第 30 行通过 switch 结构判断读入的字符，也就是*pos 的值，然后采取不同的处理方法。前三个 case 是处理控制键，分别是回车换行符及退格键。如果*pos 的值是字符\n 或\r，这表示用户键入了回车，表示命令输入结束，所以在第 34 行使*pos 为 0，也就是添加了字符串结束标识\0，然后调用 putchar 在屏幕上输出换行符\n，模拟用户的键盘动作。第 38 行是对退格键\b 的处理，第 39 行的 if 判断是阻止删除非本次输入的信息，如果没有代码"if (buf [0] != '\b')"的话，按下的退格键会将命令提示符及之前的内容删除，这就错了。pos 会减 1，指向 buf 中前一个字符，并且通过 putchar 输出\b，在屏幕上模拟删除。如果不是这些控制字符，默认就直接输出。

函数 my_shell 就是所实现的简单 shell，函数中先将当前工作目录缓存 cwd_cache 置为根目录'/'，然后通过 while 语句，循环调用 print_prompt 输出命令提示符，然后调用 readline 获取用户输入。

shell.c 的代码目前就这些，下面咱们得找个机会把 my_shell 执行，肯定是由 init 这个"祖师爷"来亲自调用啦，见代码 15-10。

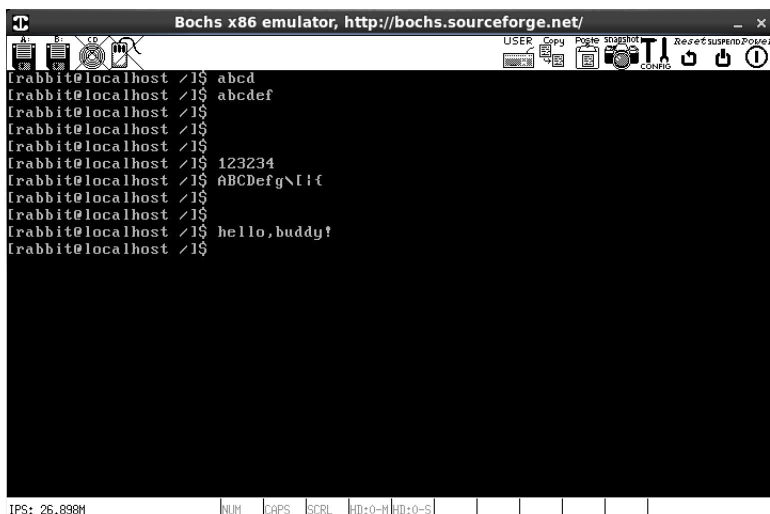
代码 15-10　（project/c15/b/kernel/main.c）

```
…略
18 int main(void) {
19     put_str("I am kernel\n");
20     init_all();
21     cls_screen();
22     console_put_str("[rabbit@localhost /]$ ");
23     while(1);
24     return 0;
```

```
25 }
26
27 /* init 进程 */
28 void init(void) {
29     uint32_t ret_pid = fork();
30     if(ret_pid) {  // 父进程
31         while(1);
32     } else {      // 子进程
33         my_shell();
34     }
35     panic("init: should not be here");
36 }
…略
```

my_shell 是在第 33 行调用的，也就是 fork 之后的子进程中。另外，在主函数中，也就是主线程执行完 init_all 之后，调用了 cls_screen 和用 console_put_str 输出了命令提示符，目的是清掉屏幕上的硬盘、分区等初始化信息。

好啦，咱们一气呵成，编译运行看结果，如图 15-5 所示。



▲图 15-5　my_shell

您看，是不是挺唬人的，似乎咱们真的拥有了一个 shell，一直看惯了挤满初始化信息的屏幕，此时突然看上去感觉好清爽，其实离最基本的 shell 还差得老远呢，无论输入什么系统都没有反应，别急，慢慢来吧。

感谢大伙儿的陪伴，下节咱们继续。

15.4.2　添加 Ctrl+u 和 Ctrl+l 快捷键

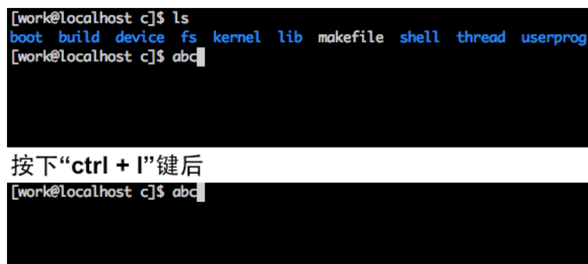
本节的目的是为演示快捷键的实现原理，如题所示，我们要实现快捷键“Ctrl+u”和“Ctrl+l”的功能。鉴于印刷后 i 和 l 容易分不清，提示一下，“Ctrl+l”中的字符'l'不是 i，其大写字符是 L。

话说在 Linux 中，“Ctrl+u”的作用是清除输入，也就是在回车前，按下“Ctrl+u”可以清掉本次的输入，相当于连续按下退格键的效果。“Ctrl+l”的作用是清屏，效果等同于 Clear 命令，但是有一点要注意，“Ctrl+l”并不会清掉本次的输入，实际效果如图 15-6 所示。

您看，在按下“Ctrl+l”快捷键后屏幕被清空了，但输入的 abc 依然还在，咱们也实现这种效果。

也许大伙儿想到，哎？在键盘驱动程序 keyboard.c 中不是可以处理按键吗？甚至我们已经实现了一些组合键，这次是否依然要在 keyboard.c 中添加？回大人，还真不是，理由如下。

（1）操作系统虽说是由中断驱动的，但中断过多的话，系统会被拖累得效率骤降。而键盘驱动程序是中断处理程序，每按下一个键就会产生两个中断（分别是通码和断码产生的中断），中断量大得惊人，为了让中断处理得快一些，咱们尽可能让中断处理程序简洁。



▲图 15-6　Linux 中“ctrl+l”快捷键的效果

（2）键盘驱动是较低层的程序，它获取的数据是最原始的数据，为了让上层程序可获得更丰富有用的信息，键盘驱动应该最大限度地保留原始数据，由上层程序决定如何处理。

回顾一下，在很久以前，我们在键盘驱动中埋下了“伏笔”，现在把“伏笔”代码贴出来给大伙儿回顾，这是代码 keyboard.c 中的部分截图，如图 15-7 所示。

```
195      /*****************  快捷键ctrl+l和ctrl+u的处理  *********************
196       * 下面是把ctrl+l和ctrl+u这两种组合键产生的字符置为：
197       * cur_char的asc码-字符a的asc码，此差值比较小，
198       * 属于asc码表中不可见的字符部分.故不会产生可见字符.
199       * 我们在shell中将ascii值为l-a和u-a的分别处理为清屏和删除输入的快捷键*/
200         if ((ctrl_down_last && cur_char == 'l') || (ctrl_down_last && cur_char == 'u')) {
201            cur_char -= 'a';
202         }
203      /****************************************************************/
204
205    /* 若kbd_buf中未满并且待加入的cur_char不为0,
206     * 则将其加入到缓冲区kbd_buf中 */
207         if (!ioq_full(&kbd_buf)) {
208            ioq_putchar(&kbd_buf, cur_char);
209         }
210         return;
```

▲图 15-7　键盘驱动中的快捷键处理

注释部分大伙儿自己看吧，我大体给您说下。

变量 cur_char 中存储的是按键的 ASCII 码，我们在 keyboard.c 的第 200 行将“ctrl+l”和“ctrl+u”组合键也转换为 ASCII 码，不过此时 cur_char 中存储的是字符 l 或字符 u 的 ASCII 码值减去字符 a 的 ASCII 码值的差。在 ASCII 码表中，ASCII 码值为十进制 0～31 和 127 的字符是控制字符，它们不可见，因此字符 l 和字符 u 的 ASCII 码值减去 a 的 ASCII 后的差会落到控制字符中，但并不是所有的控制字符都可占用，对于系统中已经处理的控制字符必须要保留。比如退格键‘\b’、换行符‘\n’和回车符‘\r’的 ASCII 码分别是 8、10 和 13，咱们已经在 shell.c 中针对它们做出了处理，因此要定义其他快捷键的话，要将这三个控制键的 ASCII 码跨过去。好啦，可以上代码啦，见代码 15-11。

代码 15-11　（project/c15/c/shell/shell.c）

```
…略
25 /* 从键盘缓冲区中最多读入 count 个字节到 buf */
26 static void readline(char* buf, int32_t count) {
…略
46              /* ctrl+l 清屏 */
47              case 'l' - 'a':
48                  /* 1 先将当前的字符'l'-'a'置为 0 */
49                  *pos = 0;
50                  /* 2 再将屏幕清空 */
51                  clear();
52                  /* 3 打印提示符 */
53                  print_prompt();
54                  /* 4 将之前键入的内容再次打印 */
55                  printf("%s", buf);
56                  break;
57
```

```
58              /* ctrl+u 清掉输入 */
59              case 'u' - 'a':
60                 while (buf != pos) {
61                    putchar('\b');
62                    *(pos--) = 0;
63                    }
64                    break;
65
66              /* 非控制键则输出字符 */
67              default:
68                 putchar(*pos);
69                 pos++;
70         }
71     }
72     printf("readline: can`t find enter_key in the cmd_line,
   max num of char is 128\n");
73 }
…略
```

对于按键的处理是由软件决定的，您可以任意处理按下的键的行为，此次更新的是 readline 函数，在第 46～56 行咱们把 ctrl+l 键处理为清屏操作，这分为四步来完成。先将 pos 指向的字符置为 0，也就是字符串结束符‘\0’。接着调用 clear 系统调用清屏，此时屏幕上空空如也。然后调用 print_prompt 函数重新输出命令提示符，也就是此时屏幕上出现了“[rabbit@localhost /]$”，最后把 buf 中的字符串，也就是用户刚刚键入的字符通过 printf 打印出来。经过这四步，我们模拟了 Linux 中的清屏快捷键“ctrl+l”的效果。

第 58～64 行是处理快捷键“ctrl+u”，其实现原理是通过 while 循环连续输出退格符，然后使指针 pos 逐步递减，并将对应位置为 0，直到 pos 指向了 buf 的起始处。

代码就介绍完了，这个例子还不太好演示，图片上不容易看出快捷键的效果，大伙儿还是自行运行一下程序检测吧。

本小节添加的快捷键仅起到抛砖引玉的作用，其他的快捷键触类旁通，大伙儿有兴趣自己添加吧。

15.4.3　解析键入的字符

上节咱们的 shell 还是个“哑”shell，无论咱们按下什么键它都没任何反应，它之所以什么都不会做的原因是它不知道用户输入的是什么。本节咱们要添加个命令解析的功能，在 shell.c 中增加函数 cmd_parse，见代码 15-12。

代码 15-12　（project/c15/d/shell/shell.c）

```
…略
 74 /* 分析字符串 cmd_str 中以 token 为分隔符的单词,
 将各单词的指针存入 argv 数组 */
 75 static int32_t cmd_parse(char* cmd_str, char** argv, char token) {
 76      assert(cmd_str != NULL);
 77      int32_t arg_idx = 0;
 78      while(arg_idx < MAX_ARG_NR) {
 79         argv[arg_idx] = NULL;
 80         arg_idx++;
 81      }
 82      char* next = cmd_str;
 83      int32_t argc = 0;
 84      /* 外层循环处理整个命令行 */
 85      while(*next) {
 86         /* 去除命令字或参数之间的空格 */
 87         while(*next == token) {
 88           next++;
 89         }
 90         /* 处理最后一个参数后接空格的情况，如"ls dir2 " */
 91         if (*next == 0) {
 92            break;
 93         }
 94         argv[argc] = next;
 95
 96         /* 内层循环处理命令行中的每个命令字及参数 */
```

```
 97          while (*next && *next != token) { // 在字符串结束前找单词分隔符
 98            next++;
 99          }
100
101          /* 如果未结束（是 token 字符），使 tocken 变成 0 */
102          if (*next) {
103              *next++ = 0;   // 将 token 字符替换为字符串结束符 0
                    // 作为一个单词的结束，并将字符指针 next 指向下一个字符
104          }
105
106          /* 避免 argv 数组访问越界，参数过多则返回 0 */
107          if (argc > MAX_ARG_NR) {
108              return -1;
109          }
110          argc++;
111    }
112    return argc;
113 }
114
115 char* argv[MAX_ARG_NR];
               // argv 必须为全局变量，为了以后 exec 的程序可访问参数
116 int32_t argc = -1;
117 /* 简单的 shell */
118 void my_shell(void) {
119     cwd_cache[0] = '/';
120     while (1) {
121        print_prompt();
122        memset(final_path, 0, MAX_PATH_LEN);
123        memset(cmd_line, 0, MAX_PATH_LEN);
124        readline(cmd_line, MAX_PATH_LEN);
125        if (cmd_line[0] == 0) {    // 若只键入了一个回车
126            continue;
127        }
128        argc = -1;
129        argc = cmd_parse(cmd_line, argv, ' ');
130        if (argc == -1) {
131            printf("num of arguments exceed %d\n", MAX_ARG_NR);
132            continue;
133        }
134
135        int32_t arg_idx = 0;
136        while(arg_idx < argc) {
137          printf("%s ", argv[arg_idx]);
138          arg_idx++;
139        }
140        printf("\n");
141    }
142    panic("my_shell: should not be here");
143 }
```

函数 cmd_parse 接受 3 个参数，用户键入的原始命令串 cmd_str、参数字符串数组 argv、分隔符 token。功能是分析字符串 cmd_str 中以 token 为分隔符的单词，将解析出来的单词的指针存入 argv 数组。

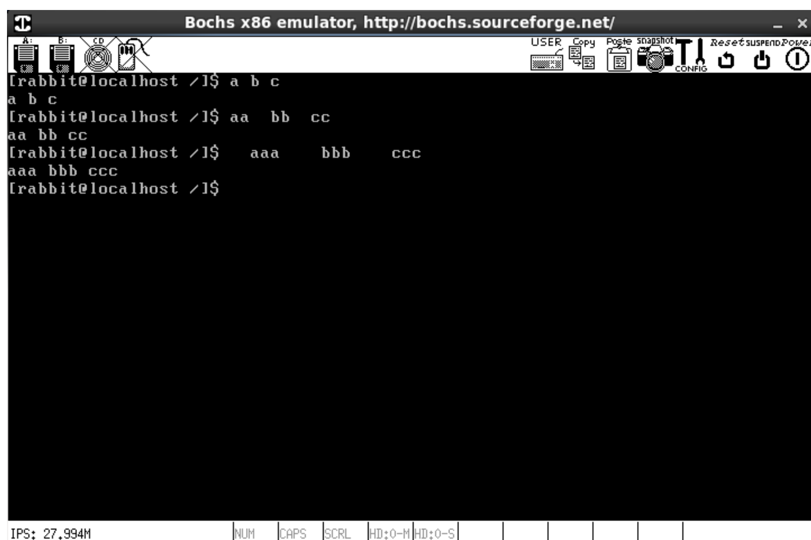
在函数开头第 78 行的 while 循环用来清空数组 argv。

第 82 行，指针 next 指向 cmd_str，next 用于处理每一个字符，第 85 行的 while 外层循环处理整个命令行 cmd_str。

在循环内部的第 1 个循环用于跨过 cmd_str 中的空格。第 91 行的 if 判断，如果 next 已经指向了 cmd_str 的结尾就退出循环，这是为了避免在最后一个参数后出现空格的情况。

第 94 行的“argv[argc] = next”，每找出一个字符串就将其在 cmd_str 中的起始 next 存储到 argv 数组。第 97 行的 while 循环用于处理命令行中的每个命令字及参数，在字符串未结束的情况下，遇到分隔符 token 后就退出，随后在第 102 行判断跳出上个 while 循环的原因是遇到了 token 字符，还是遇到了结束符 0，如果是 token 字符，则将指针 next 指向的 token 字符置为 0，也就是人为添加结束字符'\0'，使数组 argv 中的每个元素（字符串）从 cmd_str 中找到结束边界，并使 next 指向下一个字符。

下面在函数 my_shell 中添加了测试代码，第 136～139 行输出每个分离出来的字符串，好啦，运行看结果，如图 15-8 所示。



▲图 15-8　分析命令字符串

在图 15-8 中输入的各个字符串的结束都是字符 c，在其后其实都有多个空格，只是图上看不出来。另外在输出的结果中，结束字符 c 处没有空格。

好啦，本节较轻松，到这就结束了。

15.4.4　添加系统调用

到现在咱们的 shell 还名不符实，它不能帮用户做任何事情。为了让 shell 能够尽快与用户互动，咱们还需要做些基础工作，等基础设施搭建完成了，咱们的 shell 就具有“灵性”了。

在这之前咱们在 fs.c 中实现了很多系统调用的内核接口，就是那些以“sys_”开头的函数，您懂的，我们早晚会为它们添加系统调用，就是今天。

下面按照添加系统调用的三个步骤依次列出相关代码，见代码 15-13。

代码 15-13　（project/c15/e/lib/user/syscall.h）

```
…略
 6 enum SYSCALL_NR {
 7     SYS_GETPID,
 8     SYS_WRITE,
 9     SYS_MALLOC,
10     SYS_FREE,
11     SYS_FORK,
12     SYS_READ,
13     SYS_PUTCHAR,
14     SYS_CLEAR,
15     SYS_GETCWD,
16     SYS_OPEN,
17     SYS_CLOSE,
18     SYS_LSEEK,
19     SYS_UNLINK,
20     SYS_MKDIR,
21     SYS_OPENDIR,
22     SYS_CLOSEDIR,
23     SYS_CHDIR,
24     SYS_RMDIR,
25     SYS_READDIR,
26     SYS_REWINDDIR,
27     SYS_STAT,
28     SYS_PS
29 };
…略
```

以上定义的 enum SYSCALL_NR 是咱们系统中目前所支持的所有系统调用。下面是新增的系统调用

实现，除 SYS_PS 以外，其他的系统调用号对应的内核部分都已经实现了，见代码 15-14。

代码 15-14　（project/c15/e/lib/user/syscall.c）

```
…略
 92 /* 获取当前工作目录 */
 93 char* getcwd(char* buf, uint32_t size) {
 94     return (char*)_syscall2(SYS_GETCWD, buf, size);
 95 }
 96
 97 /* 以 flag 方式打开文件 pathname */
 98 int32_t open(char* pathname, uint8_t flag) {
 99     return _syscall2(SYS_OPEN, pathname, flag);
100 }
101
102 /* 关闭文件 fd */
103 int32_t close(int32_t fd) {
104     return _syscall1(SYS_CLOSE, fd);
105 }
106
107 /* 设置文件偏移量 */
108 int32_t lseek(int32_t fd, int32_t offset, uint8_t whence) {
109     return _syscall3(SYS_LSEEK, fd, offset, whence);
110 }
111
112 /* 删除文件 pathname */
113 int32_t unlink(const char* pathname) {
114     return _syscall1(SYS_UNLINK, pathname);
115 }
116
117 /* 创建目录 pathname */
118 int32_t mkdir(const char* pathname) {
119     return _syscall1(SYS_MKDIR, pathname);
120 }
121
122 /* 打开目录 name */
123 struct dir* opendir(const char* name) {
124     return (struct dir*)_syscall1(SYS_OPENDIR, name);
125 }
126
127 /* 关闭目录 dir */
128 int32_t closedir(struct dir* dir) {
129     return _syscall1(SYS_CLOSEDIR, dir);
130 }
131
132 /* 删除目录 pathname */
133 int32_t rmdir(const char* pathname) {
134     return _syscall1(SYS_RMDIR, pathname);
135 }
136
137 /* 读取目录 dir */
138 struct dir_entry* readdir(struct dir* dir) {
139     return (struct dir_entry*)_syscall1(SYS_READDIR, dir);
140 }
141
142 /* 回归目录指针 */
143 void rewinddir(struct dir* dir) {
144     _syscall1(SYS_REWINDDIR, dir);
145 }
146
147 /* 获取 path 属性到 buf 中 */
148 int32_t stat(const char* path, struct stat* buf) {
149     return _syscall2(SYS_STAT, path, buf);
150 }
151
152 /* 改变工作目录为 path */
153 int32_t chdir(const char* path) {
154     return _syscall1(SYS_CHDIR, path);
155 }
156
157 /* 显示任务列表 */
158 void ps(void) {
159     _syscall0(SYS_PS);
160 }
```

最后一个系统调用是 ps，一会儿咱们再说它，这些系统调用要在 syscall_table 中注册，见代码 15-15。

代码 15-15　（project/c15/e/userprog/syscall-init.c）

```
…略
21 /* 初始化系统调用 */
22 void syscall_init(void) {
23     put_str("syscall_init start\n");
24     syscall_table[SYS_GETPID]     = sys_getpid;
25     syscall_table[SYS_WRITE]      = sys_write;
26     syscall_table[SYS_MALLOC]     = sys_malloc;
27     syscall_table[SYS_FREE]       = sys_free;
28     syscall_table[SYS_FORK]       = sys_fork;
29     syscall_table[SYS_READ]       = sys_read;
30     syscall_table[SYS_PUTCHAR]    = sys_putchar;
31     syscall_table[SYS_CLEAR]      = cls_screen;
32     syscall_table[SYS_GETCWD]     = sys_getcwd;
33     syscall_table[SYS_OPEN]       = sys_open;
34     syscall_table[SYS_CLOSE]      = sys_close;
35     syscall_table[SYS_LSEEK]      = sys_lseek;
36     syscall_table[SYS_UNLINK]    = sys_unlink;
37     syscall_table[SYS_MKDIR]      = sys_mkdir;
38     syscall_table[SYS_OPENDIR]    = sys_opendir;
39     syscall_table[SYS_CLOSEDIR]   = sys_closedir;
40     syscall_table[SYS_CHDIR]      = sys_chdir;
41     syscall_table[SYS_RMDIR]      = sys_rmdir;
42     syscall_table[SYS_READDIR]    = sys_readdir;
43     syscall_table[SYS_REWINDDIR]  = sys_rewinddir;
44     syscall_table[SYS_STAT]       = sys_stat;
45     syscall_table[SYS_PS]        = sys_ps;
46     put_str("syscall_init done\n");
47 }…略
```

下面说一下系统调用 ps。这是为了模拟 Linux 中的 ps 命令，当然，由于能力有限，并且耐心比能力还有限，咱们的 ps 命令极其简陋，仅能打印出进程的 pid、ppid、状态、运行时间片和进程名。对应的内核部分 sys_ps 定义在 thread.c 中，见代码 15-16。

代码 15-16　（project/c15/e/thread/thread.c）

```
…略
217 /* 以填充空格的方式输出 buf */
218 static void pad_print(char* buf, int32_t buf_len, void* ptr, char format) {
219     memset(buf, 0, buf_len);
220     uint8_t out_pad_0idx = 0;
221     switch(format) {
222        case 's':
223           out_pad_0idx = sprintf(buf, "%s", ptr);
224           break;
225        case 'd':
226           out_pad_0idx = sprintf(buf, "%d", *((int16_t*)ptr));
227        case 'x':
228           out_pad_0idx = sprintf(buf, "%x", *((uint32_t*)ptr));
229     }
230     while(out_pad_0idx < buf_len) { // 以空格填充
231        buf[out_pad_0idx] = ' ';
232        out_pad_0idx++;
233     }
234     sys_write(stdout_no, buf, buf_len - 1);
235 }
236
237 /* 用于在 list_traversal 函数中的回调函数，用于针对线程队列的处理 */
238 static bool elem2thread_info(struct list_elem* pelem, int arg UNUSED) {
239     struct task_struct* pthread = \
       elem2entry(struct task_struct, all_list_tag, pelem);
240     char out_pad[16] = {0};
241
242     pad_print(out_pad, 16, &pthread->pid, 'd');
243
244     if (pthread->parent_pid == -1) {
245         pad_print(out_pad, 16, "NULL", 's');
246     } else {
```

```
247         pad_print(out_pad, 16, &pthread->parent_pid, 'd');
248     }
249
250     switch (pthread->status) {
251        case 0:
252           pad_print(out_pad, 16, "RUNNING", 's');
253           break;
254        case 1:
255           pad_print(out_pad, 16, "READY", 's');
256           break;
257        case 2:
258           pad_print(out_pad, 16, "BLOCKED", 's');
259           break;
260        case 3:
261           pad_print(out_pad, 16, "WAITING", 's');
262           break;
263        case 4:
264           pad_print(out_pad, 16, "HANGING", 's');
265           break;
266        case 5:
267           pad_print(out_pad, 16, "DIED", 's');
268     }
269     pad_print(out_pad, 16, &pthread->elapsed_ticks, 'x');
270
271     memset(out_pad, 0, 16);
272     ASSERT(strlen(pthread->name) < 17);
273     memcpy(out_pad, pthread->name, strlen(pthread->name));
274     strcat(out_pad, "\n");
275     sys_write(stdout_no, out_pad, strlen(out_pad));
276     return false; // 此处返回 false 是为了迎合主调函数 list_traversal
                      // 只有回调函数返回 false 时才会继续调用此函数
277 }
278
279 /* 打印任务列表 */
280 void sys_ps(void) {
281     char* ps_title = "PID           PPID          STAT\
           TICKS         COMMAND\n";
282     sys_write(stdout_no, ps_title, strlen(ps_title));
283     list_traversal(&thread_all_list, elem2thread_info, 0);
284 }
…略
```

代码就不详尽介绍了，下面从大体上说说。

函数 pad_print 用于对齐输出，原理是先用 switch 结构中 sprintf 函数把待输出的字符串 ptr 写入缓冲区 buf，buf 的长度是 buf_len，这是固定的值，无论字符串 ptr 是多少字符，永远输出 buf_len 长度，如果 ptr 长度不足 buf_len，就以空格来填充，这是用 while 循环完成的。在 pad_print 中的 switch 中有三种 case，case 's'用来处理字符串，case 'd'用来处理 16 位整数，case 'x'用来处理 32 位整数。本来 case 'd'和 case 'x'取其一即可，但由于待处理的数据类型不同，pid 是 16 位宽，若统一用 32 位来处理，还需要再改 pid 的数据类型，改个类型都不叫事儿，恐怖的是与之产生的“雪崩效应”，我还得改书中所有涉及到 pid 的代码和相关的解释内容，还包括在这之后的所有章节中与此相关的代码，话说我已经改伤了，过程实在是太痛苦了，很对不起大家，因此为 pid 专门指定了 case 'd'，使其处理 16 位数据，解释这个的目的是想和大伙儿说，如果当初 task struct 中的 pid 为 32 位数据类型，此处只需要处理 32 位数据就行了，不需要增加一种 case 来分开处理，请大伙包容或无视这个“补丁”。

函数 elem2thread_info 用于打印任务信息，它是 list_traversal 函数中的回调函数，用于线程队列的处理。

函数原理是输出每个任务的 pid、ppid，然后通过 switch 结构根据任务的 status 输出不同的任务状态，任务状态包括“RUNNING”“READY”“BLOCKED”“WAITING”“HANGING”“DIED”。调用 pad_print 函数把输出的信息对齐为 16 个字符的固定长度，然后通过 sys_write 输出。

最后是函数 sys_ps，它就是系统调用 ps 的内核部分。

有关系统调用的部分就到这了，基本上对于目前的应用来说已经够用了，本节到这结束。

15.4.5　路径解析转换

下面咱们进行基础工作的剩余部分——路径解析。两个问题：什么是路径解析？为什么要进行路径解析？

归根结底的原因是为了使用户操作方便，路径有绝对路径和相对路径之分。

通常情况下，用户为了方便操作，在输入命令或参数时，往往都输入的是相对路径，很少有同学老老实实地写绝对路径，毕竟太长了，输入起来好麻烦。“相对路径”虽然是用户提出的需求，但根本上是系统很贴心，它提供了相对路径的功能，这样用户才方便。相对路径就是以当前工作路径为基础，命令或参数都是相对于当前工作路径得出的，“当前工作路径”+“相对路径”=“绝对路径”。比如当前工作路径是“/home/work”，若键入了“file1”，file1 的绝对路径就是“/home/work/file1”。Linux 操作中用的最多的相对目录应该就是“.”和“..”了，这也是相对路径的表示方式，它们分别表示当前工作目录和当前工作目录的父目录。系统是如何区分相对目录和绝对目录呢？很简单，如果输入的路径以“根”目录“/”开头，顾名思义，“根”嘛，这必然是绝对目录，如果路径并非以表示根目录的“/”开头，就被“认为”是相对路径，这里强调的是“认为”，也就是会按照相对路径来处理，它未必是真正的相对路径，毕竟相对路径也得是文件系统上存在的路径，如果该路径不存在，系统会报错的。比如，用户可以执行命令“ls　file1”，如果 file1 在当前目录中存在的话，系统才会显示文件 file1 的信息。

路径输入发生在用户态，而系统调用通过中断的方式发生在内核态，咱们这里反复强调的一句话是操作系统虽是中断驱动的，但我们又希望它不停地运行，故不希望执行中断处理程序的时间过长，因此我们要为内核代码减荷，让它们尽量快点从内核态返回，以处理更多的中断。于是很自然地想到，我们不应该把路径转换的工作交给内核态下的文件系统函数，最好由用户态的程序完成，提交给内核态下文件系统函数的路径参数应该是由用户态程序转换后的绝对路径。

以上的内容已经回答了本节开头的两个问题。另外，有些命令有默认的参数，也就是在不输入参数的情况下命令也会正确执行，只是会自动执行某种特定行为。举两个例子，直接输入 cd 命令后不接参数，默认情况下是回到用户的家目录。ls 命令若无参数，则会显示当前目录下的所有文件，此文件是指一切皆文件，并非特指普通文件。

好啦，以上就是咱们本节要完成的任务，不过大伙儿放心，咱们的作风一贯是简单，路径解析也主要是把路径中的“..”和“.”替换成实际的目录，将用户键入的路径，无论是绝对路径，还是相对路径，一律转换成不含“.”和“..”的绝对路径，然后再把转换后的路径作为命令的参数，最后再对某些有默认参数的命令做针对性处理就好了。

有关路径解析的代码咱们放在 shell/buildin_cmd.c 中，这是新创建的文件，您看名字就猜到了，这是要实现内建命令的节奏，不过这以后再说，还是先做好基础构建，上代码啦，见代码 15-17。

代码 15-17　（project/c15/e/shell/buildin_cmd.c）

```
…略
11 /* 将路径 old_abs_path 中的..和.转换为实际路径后存入 new_abs_path */
12 static void wash_path(char* old_abs_path, char* new_abs_path) {
13     assert(old_abs_path[0] == '/');
14     char name[MAX_FILE_NAME_LEN] = {0};
15     char* sub_path = old_abs_path;
16     sub_path = path_parse(sub_path, name);
17     if (name[0] == 0) {
                 // 若只键入了"/",直接将"/"存入 new_abs_path 后返回
18         new_abs_path[0] = '/';
19         new_abs_path[1] = 0;
20         return;
21     }
22     new_abs_path[0] = 0;    // 避免传给 new_abs_path 的缓冲区不干净
23     strcat(new_abs_path, "/");
24     while (name[0]) {
25        /* 如果是上一级目录“..” */
26        if (!strcmp("..", name)) {
27              char* slash_ptr =  strrchr(new_abs_path, '/');
28     /*如果未到 new_abs_path 中的顶层目录，就将最右边的'/'替换为 0，
29     这样便去除了 new_abs_path 中最后一层路径，相当于到了上一级目录 */
30        if (slash_ptr != new_abs_path) {
     // 如 new_abs_path 为“/a/b”，".."之后则变为“/a”
31             *slash_ptr = 0;
32        } else {    // 如 new_abs_path 为"/a"，".."之后则变为"/"
```

```
33      /* 若 new_abs_path 中只有 1 个'/'，即表示已经到了顶层目录，
34          就将下一个字符置为结束符 0 */
35             *(slash_ptr + 1) = 0;
36          }
37       } else if (strcmp(".", name)) {
        // 如果路径不是‘.’，就将 name 拼接到 new_abs_path
38          if (strcmp(new_abs_path, "/")) {  // 如果 new_abs_path 不是"/"
        // 就拼接一个"/",此处的判断是为了避免路径开头变成这样"//"
39               strcat(new_abs_path, "/");
40          }
41          strcat(new_abs_path, name);
42       }  // 若 name 为当前目录"."，无需处理 new_abs_path
43
44       /* 继续遍历下一层路径 */
45       memset(name, 0, MAX_FILE_NAME_LEN);
46       if (sub_path) {
47           sub_path = path_parse(sub_path, name);
48       }
49    }
50 }
51
52   /* 将 path 处理成不含..和.的绝对路径，存储在 final_path */
53 void make_clear_abs_path(char* path, char* final_path) {
54      char abs_path[MAX_PATH_LEN] = {0};
55      /* 先判断是否输入的是绝对路径 */
56      if (path[0] != '/') {      // 若输入的不是绝对路径，就拼接成绝对路径
57          memset(abs_path, 0, MAX_PATH_LEN);
58          if (getcwd(abs_path, MAX_PATH_LEN) != NULL) {
59              if (!((abs_path[0] == '/') && (abs_path[1] == 0))) {
        // 若 abs_path 表示的当前目录不是根目录/
60                  strcat(abs_path, "/");
61          }
62       }
63    }
64    strcat(abs_path, path);
65    wash_path(abs_path, final_path);
66 }
…略
```

函数 wash_path 接受两个参数，转换前的旧绝对路径 old_abs_path 和转换后的新绝对路径 new_abs_path，功能是将路径 old_abs_path 中的“..”和“.”转换为实际路径后存入 new_abs_path。其中 old_abs_path 肯定是绝对目录，这是由主调函数传入的，因此 new_abs_path 必然也是绝对路径，这两者的区别就是 old_abs_path 中可能包括“.”或“..”，但 new_abs_path 中绝对不包括它们。

wash_path 的原理是调用函数 path_parse 从左到右解析 old_abs_path 路径中的每一层，若解析出来的目录名不是“..”，就将其连接到 new_abs_path，若是“..”，就将 new_abs_path 的最后一层目录去掉。强调一下，new_abs_path 才是转换后的绝对路径的结果，在路径解析中遇到“..”时就是去修改 new_abs_path。

函数开头定义了数组 name[MAX_FILE_NAME_LEN]，用它来存储路径中解析出来的各层目录名。第 15 行指针 sub_path 指向 old_abs_path，用它来配合函数 path_parse 的路径解析。

name 数组本身初始化为 0，它就是空的，在经过 path_parse 处理后，什么情况下 name 依然为空呢？如果 old_abs_path 本身为空，name 并未改变，因此依然为空，不过函数开头的 assert 就会报警，后面的代码不会执行。如果 old_abs_path 仅由一个或连续多个“/”组成，path_parse 会将这些“/”去掉，此时数组 name 依然为空。如果 old_abs_path 不是单纯的“/”，且不为空，经过 path_parse 返回后，name 必然不为空。因此在第 17 行，如果 name[0]=0，即 name 为空，一定是 old_abs_path 仅为 1 个以上的“/”，此时把它当根目录处理，将 new_abs_path 填充为根目录“/”后返回。

第 23 行人为在 new_path 后接一个“/”，作为路径分隔符。

第 26 行，如果解析出来的目录是“..”，按照语义就将目录回退到上一级目录。先用函数 strrchr 从右往左找到 new_abs_path 中最右边的“/”的地址，注意是地址，并不是下标，该地址用指针 slash_ptr 保存。

第 30～36 行判断 slash_ptr 是否是 new_abs_path 的首字符的地址，如果不是，就将 slash_ptr 处的值置为字符串结束字符“\0”，也就是 0 值，否则就将 slash_ptr 处的下一个字符置为结束符“\0”。是这样的，

如果最后一个“/”不是 new_abs_path 首字符的“/”，也就是说 new_abs_path 是个多级目录，其中不只有根目录的“/”，类似这种情况：“/a/b”，slash_ptr 是 a 和 b 之间的“/”，将最右一个“/”替换为 0，这样便去除了 new_abs_path 中最后一层路径，相当于到了上一级目录，即 new_abs_path 变成了“/a”。如果 slash_ptr 处的“/”是 new_abs_path 的首字符“/”，类似这种情况：“/a”，就将字符 a 变成 0，new_abs_path 变成了“/”。

第 37 行，如果解析出来的目录名 name 不是“.”，就将目录名 name 拼接到 new_abs_path 的后面。如果目录名是“.”，就什么都不做，保持 new_abs_path 不变。

第 44～48 行继续下一轮处理，本函数介绍完了。

函数 make_clear_abs_path 接受 2 个参数，原目录 path 和转换后的绝对路径 final_path。其中 path 是用户键入的路径，可能是相对路径，也可能是绝对路径，也可能包含“.”和“..”的相对路径或绝对路径，而 final_path 是不含“.”和“..”的绝对路径。

函数实现就是上面介绍过的 wash_path 的封装，就不那么详细地介绍了。核心思路是：为了确保传给 wash_path 的参数 old_abs_path 是绝对路径，在第 58 行调用了系统调用 getcwd 获得当前工作目录的绝对路径，将用户输入的目录 path 追加到工作目录之后形成绝对目录 abs_path，将其作为参数传给 wash_path 进行目录转换。

下面在 shell.c 中增加测试代码，如代码 15-18 所示。

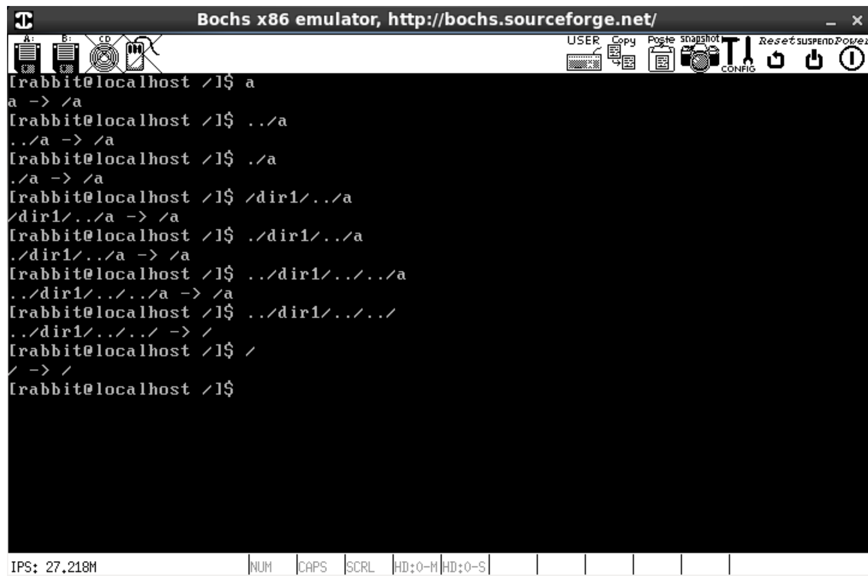
代码 15-18　（project/c15/e/shell/shell.c）

```
…略
116 char* argv[MAX_ARG_NR];
                    // argv 为全局变量，为了以后 exec 的程序可访问参数
117 int32_t argc = -1;
118 /* 简单的 shell */
119 void my_shell(void) {
120     cwd_cache[0] = '/';
121     cwd_cache[1] = 0;
122     while (1) {
123        print_prompt();
124        memset(final_path, 0, MAX_PATH_LEN);
125        memset(cmd_line, 0, MAX_PATH_LEN);
126        readline(cmd_line, MAX_PATH_LEN);
127        if (cmd_line[0] == 0) {    // 若只键入了一个回车
128            continue;
129        }
130        argc = -1;
131        argc = cmd_parse(cmd_line, argv, ' ');
132        if (argc == -1) {
133            printf("num of arguments exceed %d\n", MAX_ARG_NR);
134            continue;
135        }
136
137        char buf[MAX_PATH_LEN] = {0};
138        int32_t arg_idx = 0;
139        while(arg_idx < argc) {
140           make_clear_abs_path(argv[arg_idx], buf);
141           printf("%s -> %s\n", argv[arg_idx], buf);
142           arg_idx++;
143        }
144    }
145    panic("my_shell: should not be here");
146 }
```

第 139～143 行是本次的测试代码，循环打印所键入的原始路径字符串和转换后的路径。运行结果如图 15-9 所示。

当前工作目录是根目录，目测图中所键入的路径均被转换为正确的绝对路径。

好啦，到这该说的都说啦，估计兄弟们都已经厌倦了基础工作，好吧，下节咱们开始让 shell 动起来，大伙儿辛苦了，下节再见。



▲图 15-9　路径转换演示

15.4.6　实现 ls、cd、mkdir、ps、rm 等命令

如果您接触过 Linux 的 shell 或微软的 DoS，就一定了解命令分为两大类，一种是外部命令，另一种是内部命令。

外部命令是指该命令是个存储在文件系统上的外部程序，执行该命令实际上是从文件系统上加载该程序到内存后运行的过程，也就是说外部命令会以进程的方式执行。大伙儿应该最为熟悉 ls 命令，它就是典型的外部命令，它通常的存储路径是/bin/ls。

内部命令也称为内建命令，是系统本身提供的功能，它们并不以单独的程序文件存在，只是一些单独的功能函数，系统执行这些命令实际上是在调用这些函数。比如 cd、fg、jobs 等命令是由 bash 提供的，因此它们称为 BASH_BUILTINS。

为了让咱们的 shell 动起来，本节咱们要实现一些命令，这些命令包括 ls、cd、mkdir、rmdir、rm、pwd、ps 和 clear。注意啦，虽然这些命令在 Linux 中大部分都属于外部命令，但这并不影响 shell 功能的实现，为了省事，咱们目前统统用内部函数的方式来实现它们，先让 shell 动起来再说。

我们的内部命令是在 shell/buildin_cmd.c 文件中，见代码 15-19。

代码 15-19　（project/c15/f/shell/buildin_cmd.c）

```
…略
 68 /* pwd 命令的内建函数 */
 69 void buildin_pwd(uint32_t argc, char** argv UNUSED) {
 70     if (argc != 1) {
 71         printf("pwd: no argument support!\n");
 72         return;
 73     } else {
 74         if (NULL != getcwd(final_path, MAX_PATH_LEN)) {
 75             printf("%s\n", final_path);
 76         } else {
 77             printf("pwd: get current work directory failed.\n");
 78         }
 79     }
 80 }
 81
 82 /* cd 命令的内建函数 */
 83 char* buildin_cd(uint32_t argc, char** argv) {
 84     if (argc > 2) {
 85         printf("cd: only support 1 argument!\n");
```

```
 86         return NULL;
 87     }
 88
 89     /* 若是只键入 cd 而无参数，直接返回到根目录 */
 90     if (argc == 1) {
 91         final_path[0] = '/';
 92         final_path[1] = 0;
 93     } else {
 94         make_clear_abs_path(argv[1], final_path);
 95     }
 96
 97     if (chdir(final_path) == -1) {
 98          printf("cd: no such directory %s\n", final_path);
 99          return NULL;
100     }
101     return final_path;
102 }
103
104 /* ls 命令的内建函数 */
105 void buildin_ls(uint32_t argc, char** argv) {
106     char* pathname = NULL;
107     struct stat file_stat;
108     memset(&file_stat, 0, sizeof(struct stat));
109     bool long_info = false;
110     uint32_t arg_path_nr = 0;
111     uint32_t arg_idx = 1;   // 跨过 argv[0]，argv[0]是字符串“ls”
112     while (arg_idx < argc) {
113        if (argv[arg_idx][0] == '-') {      // 如果是选项，单词的首字符是-
114            if (!strcmp("-l", argv[arg_idx])) {        // 如果是参数-l
115                 long_info = true;
116            } else if (!strcmp("-h", argv[arg_idx])) {   // 参数-h
117                printf("usage: -l list all infomation about the file.\n-h for \
           help\nlist all files in the current dirctory if no option\n");
118                return;
119            } else {      // 只支持-h -l 两个选项
120                printf("ls: invalid option %s\nTry `ls -h' for more\
                   information.\n", argv[arg_idx]);
121                return;
122            }
123        } else {       // ls 的路径参数
124            if (arg_path_nr == 0) {
125                pathname = argv[arg_idx];
126                arg_path_nr = 1;
127            } else {
128                printf("ls: only support one path\n");
129                return;
130            }
131        }
132        arg_idx++;
133     }
134
135     if (pathname == NULL) {       // 若只输入了 ls 或 ls -l
             // 没有输入操作路径，默认以当前路径的绝对路径为参数
136         if (NULL != getcwd(final_path, MAX_PATH_LEN)) {
137             pathname = final_path;
138         } else {
139             printf("ls: getcwd for default path failed\n");
140             return;
141         }
142     } else {
143         make_clear_abs_path(pathname, final_path);
144         pathname = final_path;
145     }
146
147     if (stat(pathname, &file_stat) == -1) {
148          printf("ls: cannot access %s: No such file or directory\n", pathname);
149          return;
150     }
151     if (file_stat.st_filetype == FT_DIRECTORY) {
152          struct dir* dir = opendir(pathname);
153          struct dir_entry* dir_e = NULL;
154          char sub_pathname[MAX_PATH_LEN] = {0};
```

```
155         uint32_t pathname_len = strlen(pathname);
156         uint32_t last_char_idx = pathname_len - 1;
157         memcpy(sub_pathname, pathname, pathname_len);
158         if (sub_pathname[last_char_idx] != '/') {
159              sub_pathname[pathname_len] = '/';
160              pathname_len++;
161         }
162         rewinddir(dir);
163         if (long_info) {
164              char ftype;
165              printf("total: %d\n", file_stat.st_size);
166              while((dir_e = readdir(dir))) {
167                 ftype = 'd';
168                 if (dir_e->f_type == FT_REGULAR) {
169                      ftype = '-';
170                 }
171                 sub_pathname[pathname_len] = 0;
172                 strcat(sub_pathname, dir_e->filename);
173                 memset(&file_stat, 0, sizeof(struct stat));
174                 if (stat(sub_pathname, &file_stat) == -1) {
175                      printf("ls: cannot access %s: No such file or directory\n",\
                         dir_e->filename);
176                      return;
177                 }
178                 printf("%c  %d  %d  %s\n", ftype, \ dir_e->i_no, file_stat.st_size,
                 dir_e->filename);
179             }
180        } else {
181            while((dir_e = readdir(dir))) {
182               printf("%s ", dir_e->filename);
183            }
184            printf("\n");
185        }
186        closedir(dir);
187     } else {
188        if (long_info) {
189             printf("-  %d  %d  %s\n",file_stat.st_ino,\file_stat.st_size,pathname);
190        } else {
191            printf("%s\n", pathname);
192        }
193     }
194 }
195
196   /* ps 命令内建函数 */
197   void buildin_ps(uint32_t argc, char** argv UNUSED) {
198     if (argc != 1) {
199          printf("ps: no argument support!\n");
200          return;
201     }
202     ps();
203 }
204
205   /* clear 命令内建函数 */
206   void buildin_clear(uint32_t argc, char** argv UNUSED) {
207     if (argc != 1) {
208          printf("clear: no argument support!\n");
209          return;
210     }
211     clear();
212 }
213
214   /* mkdir 命令内建函数 */
215   int32_t buildin_mkdir(uint32_t argc, char** argv) {
216     int32_t ret = -1;
217     if (argc != 2) {
218          printf("mkdir: only support 1 argument!\n");
219     } else {
220         make_clear_abs_path(argv[1], final_path);
221         /* 若创建的不是根目录 */
222         if (strcmp("/", final_path)) {
223             if (mkdir(final_path) == 0) {
224                  ret = 0;
```

```
225             } else {
226                 printf("mkdir: create directory %s failed.\n", argv[1]);
227             }
228         }
229     }
230     return ret;
231 }
232
233    /* rmdir 命令内建函数 */
234    int32_t buildin_rmdir(uint32_t argc, char** argv) {
235     int32_t ret = -1;
236     if (argc != 2) {
237         printf("rmdir: only support 1 argument!\n");
238     } else {
239         make_clear_abs_path(argv[1], final_path);
240     /* 若删除的不是根目录 */
241         if (strcmp("/", final_path)) {
242             if (rmdir(final_path) == 0) {
243                 ret = 0;
244             } else {
245                 printf("rmdir: remove %s failed.\n", argv[1]);
246             }
247         }
248     }
249     return ret;
250 }
251
252    /* rm 命令内建函数 */
253    int32_t buildin_rm(uint32_t argc, char** argv) {
254     int32_t ret = -1;
255     if (argc != 2) {
256         printf("rm: only support 1 argument!\n");
257     } else {
258         make_clear_abs_path(argv[1], final_path);
259     /* 若删除的不是根目录 */
260         if (strcmp("/", final_path)) {
261             if (unlink(final_path) == 0) {
262                 ret = 0;
263         } else {
264             printf("rm: delete %s failed.\n", argv[1]);
265         }
266
267         }
268     }
269     return ret;
270 }
```

代码很长，没有将它们拆分，原因是没必要细致介绍它们，因为这些内建命令只属于应用程序编程，就像大伙儿平时在宿主系统上开发 C 程序是一样的，大伙儿一定能轻松搞定。说下咱们内部命令的编写规则。

（1）内部命令都以前缀“buildin_”+“命令名”的形式命名，如 cd 命令的函数是 buildin_cd。

（2）形参均是 argc 和 argv，argc 是参数数组 argv 中参数的个数。

（3）函数实现是调用同功能的系统调用实现的，如函数 buildin_cd 是调用系统调用 chdir 完成的。

（4）在进行系统调用前调用函数 make_clear_abs_path 把路径转换为绝对路径。

所有内部命令的函数实现都是按照以上规则编写的，代码大伙儿自己看吧。这里稍微提一下，Linux 中执行 ls 命令时，默认是不会显示“.”和“..”的，它们被隐藏了，通过参数-a 才会显示它们。咱们的 ls 没做那么复杂，目前只支持-h 和-l 参数，“.”和“..”总会显示。

也许您在想，这些内部命令该如何调用？这肯定是在 shell.c 中完成的，shell 知道用户键入了什么，因此它知道该调用什么样的命令，好啦，下面看最新的 shell.c，它今天长这样，见代码 15-20。

代码 15-20　（project/c15/f/shell/shell.c）

```
…略
 13 /* 存储输入的命令 */
 14 static char cmd_line[MAX_PATH_LEN] = {0};
 15 char final_path[MAX_PATH_LEN] = {0};      // 用于清洗路径时的缓冲
 16
```

```
 17 /* 用来记录当前目录，是当前目录的缓存，
    每次执行 cd 命令时会更新此内容 */
 18 char cwd_cache[MAX_PATH_LEN] = {0};
 19
 20 /* 输出提示符 */
 21 void print_prompt(void) {
 22     printf("[rabbit@localhost %s]$ ", cwd_cache);
 23 }
…略
107 char* argv[MAX_ARG_NR];
               // argv 必须为全局变量，为了以后 exec 的程序可访问参数
108 int32_t argc = -1;
109 /* 简单的 shell */
110 void my_shell(void) {
111     cwd_cache[0] = '/';
112     while (1) {
113         print_prompt();
114         memset(final_path, 0, MAX_PATH_LEN);
115         memset(cmd_line, 0, MAX_PATH_LEN);
116         readline(cmd_line, MAX_PATH_LEN);
117         if (cmd_line[0] == 0) {    // 若只键入了一个回车
118             continue;
119         }
120         argc = -1;
121         argc = cmd_parse(cmd_line, argv, ' ');
122         if (argc == -1) {
123             printf("num of arguments exceed %d\n", MAX_ARG_NR);
124             continue;
125         }
126         if (!strcmp("ls", argv[0])) {
127             buildin_ls(argc, argv);
128         } else if (!strcmp("cd", argv[0])) {
129             if (buildin_cd(argc, argv) != NULL) {
130                 memset(cwd_cache, 0, MAX_PATH_LEN);
131                 strcpy(cwd_cache, final_path);
132             }
133         } else if (!strcmp("pwd", argv[0])) {
134             buildin_pwd(argc, argv);
135         } else if (!strcmp("ps", argv[0])) {
136             buildin_ps(argc, argv);
137         } else if (!strcmp("clear", argv[0])) {
138             buildin_clear(argc, argv);
139         } else if (!strcmp("mkdir", argv[0])){
140             buildin_mkdir(argc, argv);
141         } else if (!strcmp("rmdir", argv[0])){
142             buildin_rmdir(argc, argv);
143         } else if (!strcmp("rm", argv[0])) {
144             buildin_rm(argc, argv);
145         } else {
146             printf("external command\n");
147         }
148     }
149     panic("my_shell: should not be here");
150 }
…略
```

第 15 行定义的 final_path 是全局数组，它用于存储路径清洗转换的结果，同时它也是个全局缓冲区，所有内部函数都可以使用它。咱们只支持单控制台，因此并不会出现 final_path 被覆盖的情况。

代码第 126～147 行是 shell 调用内部命令的代码，核心思路是 argv[0]被认为是命令，然后调用 strcmp 函数与函数名比较，若相等则调用相应的 buildin_函数。这里要说下 cd 命令的处理，cd 命令会改变当前工作目录，当前工作目录也会在命令提示符中显示，因此 cd 命令成功执行后，要将最新的路径 final_path 复制到 cwd_cache，以更新命令提示符中的当前工作路径部分。

好啦，想必大伙儿尚未看完代码，就已经等不及想看看运行效果了对不对，不是的话就当我自娱自乐了，运行结果如图 15-10 所示。

限于屏幕大小有限，这里只运行了几个命令，大伙儿自己看下吧。本节到这也就结束了，咱们下节再见。

```
Bochs x86 emulator, http://bochs.sourceforge.net/
[rabbit@localhost /]$ ls -l
total: 96
d  0  96   .
d  0  96   ..
-  1  24   file1
d  2  48   dir1
[rabbit@localhost /]$ ps
PID            PPID           STAT           TICKS          COMMAND
1              NULL           READY          292960         init
2              NULL           READY          292960         main
3              NULL           BLOCKED        0              idle
4              1              RUNNING        93             init_fork
[rabbit@localhost /]$ cd dir
cd: no such directory /dir
[rabbit@localhost /]$ cd dir1
[rabbit@localhost /dir1]$ ls
. ..
[rabbit@localhost /dir1]$ mkdir dir11
[rabbit@localhost /dir1]$ mkdir dir11/dir111
[rabbit@localhost /dir1]$ ls dir11
. .. dir111
[rabbit@localhost /dir1]$ cd dir11/dir111
[rabbit@localhost /dir1/dir11/dir111]$ pwd
/dir1/dir11/dir111
[rabbit@localhost /dir1/dir11/dir111]$ _
CTRL + 3rd button enables mouse  NUM  CAPS  SCRL  HD:0-M  HD:0-S
```

▲图 15-10　shell 响应命令

15.5 加载用户进程

上节中，咱们的 shell 已经可以支持内部命令了，但这还远远不够，我们必须要做到从硬盘上加载程序，实现真正的进程。

15.5.1　实现 exec

大伙儿早，为了进一步完善用户进程，咱们今天要完成 exec 的实现。两个问题：第一它是干吗的？第二为什么要实现 exec？

先回答第一个，exec 会把一个可执行文件的绝对路径作为参数，把当前正在运行的用户进程的进程体（代码段、数据段、堆、栈）用该可执行文件的进程体替换，从而实现了新进程的执行。注意，exec 只是用新进程的进程体替换老进程进程体，因此新进程的 pid 依然是老进程 pid。

下面解释下第二个问题。

我们在上节中虽然实现了一些内部命令，但显然那种方法太笨拙了，我们是利用一系列的“if-else if”来完成的。您看，之所以能够用“if-else if”结构来实现命令处理，原因是我们能够提前预见用户要键入什么样的命令串，抱歉，与其说是“预见”，不如说是“限制”，实际上用户只能键入“if-else if”结构中包含的命令。显然，如果按照这种笨拙的方法继续添加新命令，工作量大不说，难道每支持一个新命令就要重新编译一次 shell 不成？最要命的是外部命令都是存储在文件系统上的外部程序，程序名可自由命名，现有的“if-else if”结构根本无法预见程序名是什么，因此如果用户想运行一个外部程序就没办法了。有了 exec，用户便可以完成任意外部命令（用户进程）的运行。

```
NAME
       execl, execlp, execle, execv, execvp - execute a file

SYNOPSIS
       #include <unistd.h>

       extern char **environ;

       int execl(const char *path, const char *arg, ...);
       int execlp(const char *file, const char *arg, ...);
       int execle(const char *path, const char *arg,
                  ..., char * const envp[]);
       int execv(const char *path, char *const argv[]);
       int execvp(const char *file, char *const argv[]);
```

▲图 15-11　exec 函数族

exec 是个簇，簇中包括的函数如图 15-11 所示。

图中这五个 exec 函数功能类似，差别在于程序对象的表示方式和是否传入环境变量。参数 path 表示是可执行文件绝对路径，参数 file 表示可执行程序名，具体的路

径将从 shell 的全局变量$PATH 中指定的路径中搜索。参数…表示可变参数，其中包括参数及环境变量。

咱们就不细说了，因为我们并不会都实现，就挑个简单的来完成吧，这里选的是 execv，execv 失败后则返回−1，成功则无返回值。这个看上去有点奇怪是吧，成功了怎么不返回 0？原因是：exec 是去执行一新进程，相当于 jmp 指令一去不回头，并不会“返回”，此函数没必要有返回值。既然 exec 是执行新进程，那总该知道进程是否运行成功吧？有没有办法获得进程的返回值？既然说了就表示当然有办法，以后咱们会介绍。

exec 函数定义在 userprog/exec.c 中，exec.c 是本节新增的文件，见代码 15-21-1。

代码 15-21-1　（project/c15/g/userprog/exec.c）

```
…略
 9 extern void intr_exit(void);
10 typedef uint32_t Elf32_Word, Elf32_Addr, Elf32_Off;
11 typedef uint16_t Elf32_Half;
12
13 /* 32 位 elf 头 */
14 struct Elf32_Ehdr {
15     unsigned char e_ident[16];
16     Elf32_Half    e_type;
17     Elf32_Half    e_machine;
18     Elf32_Word    e_version;
19     Elf32_Addr    e_entry;
20     Elf32_Off     e_phoff;
21     Elf32_Off     e_shoff;
22     Elf32_Word    e_flags;
23     Elf32_Half    e_ehsize;
24     Elf32_Half    e_phentsize;
25     Elf32_Half    e_phnum;
26     Elf32_Half    e_shentsize;
27     Elf32_Half    e_shnum;
28     Elf32_Half    e_shstrndx;
29 };
30
31 /* 程序头表 Program header 就是段描述头 */
32 struct Elf32_Phdr {
33     Elf32_Word p_type;          // 见下面的 enum segment_type
34     Elf32_Off  p_offset;
35     Elf32_Addr p_vaddr;
36     Elf32_Addr p_paddr;
37     Elf32_Word p_filesz;
38     Elf32_Word p_memsz;
39     Elf32_Word p_flags;
40     Elf32_Word p_align;
41 };
42
43 /* 段类型 */
44 enum segment_type {
45     PT_NULL,           // 忽略
46     PT_LOAD,           // 可加载程序段
47     PT_DYNAMIC,        // 动态加载信息
48     PT_INTERP,         // 动态加载器名称
49     PT_NOTE,           // 一些辅助信息
50     PT_SHLIB,          // 保留
51     PT_PHDR            // 程序头表
52 };
…略
```

代码 15-21-1 中定义了 elf 相关的数据结构。咱们的用户进程是用 C 语言编写的，编译为 elf 格式，因此要把用户程序从文件系统上加载到内存执行，必然涉及到 elf 格式的解析。其实早在加载内核时咱们已经接触过 elf 文件解析工作了，只不过那时更困难一些，用的是汇编代码，现在咱们用 C 语言来做同样的工作。既然更困难的时期都熬过来了，眼前的这点都不叫事儿。我们在第 5 章介绍了 elf 格式的内容，下面的加载工作很大一部分就是分析 elf 中的各种头，大伙儿忘了可以回头翻翻再继续。

第 10～11 行定义了一些以前缀 Elf32_开头的变量，这是为了在“名称”上与 elf 相关结构中的变量类型吻合，其实变量类型只是存储数值的空间大小而已，ELF 结构字段中的变量大小分别是 4 字节和 2 字节。

结构体 struct Elf32_Ehdr 定义的是 32 位 elf 文件头。接下来是结构体 struct Elf32_Phdr，它表示程序头

表，也就是段头表。枚举类型 enum segment_type 表示可识别的段的类型，这里咱们只关注类型为 PT_LOAD 的段就可以了，它是可加载的段，也就是程序本身的程序体。以上各结构的详细内容都可以在第 5 章中 elf 小节中找到，不再赘述。下面看代码 15-21-2。

代码 15-21-2　（project/c15/g/userprog/exec.c）

```
…略
54 /* 将文件描述符 fd 指向的文件中，偏移为 offset，
   大小为 filesz 的段加载到虚拟地址为 vaddr 的内存 */
55 static bool segment_load(int32_t fd, uint32_t offset, \
      uint32_t filesz, uint32_t vaddr) {
56      uint32_t vaddr_first_page = vaddr & 0xfffff000;
      // vaddr 地址所在的页框
57      uint32_t size_in_first_page = PG_SIZE - (vaddr & 0x00000fff);
      // 加载到内存后，文件在第一个页框中占用的字节大小
58      uint32_t occupy_pages = 0;
59      /* 若一个页框容不下该段 */
60      if (filesz > size_in_first_page) {
61          uint32_t left_size = filesz - size_in_first_page;
62          occupy_pages = DIV_ROUND_UP(left_size, PG_SIZE) + 1;
           // 1 是指 vaddr_first_page
63      } else {
64          occupy_pages = 1;
65      }
66
67      /* 为进程分配内存 */
68      uint32_t page_idx = 0;
69      uint32_t vaddr_page = vaddr_first_page;
70      while (page_idx < occupy_pages) {
71         uint32_t* pde = pde_ptr(vaddr_page);
72         uint32_t* pte = pte_ptr(vaddr_page);
73
74         /* 如果 pde 不存在，或者 pte 不存在就分配内存.
75          * pde 的判断要在 pte 之前，否则 pde 若不存在会导致
76          * 判断 pte 时缺页异常 */
77        if (!(*pde & 0x00000001) || !(*pte & 0x00000001)) {
78             if (get_a_page(PF_USER, vaddr_page) == NULL) {
79                  return false;
80             }
81        } // 如果原进程的页表已经分配了，利用现有的物理页
          // 直接覆盖进程体
82         vaddr_page += PG_SIZE;
83         page_idx++;
84     }
85     sys_lseek(fd, offset, SEEK_SET);
86     sys_read(fd, (void*)vaddr, filesz);
87     return true;
88 }
…略
```

函数 segment_load 接受 4 个参数，文件描述符 fd、段在文件中的字节偏移量 offset、段大小 filesz、段被加载到的虚拟地址 vaddr，函数功能是将文件描述符 fd 指向的文件中，偏移为 offset，大小为 filesz 的段加载到虚拟地址为 vaddr 的内存空间。也许您会感到奇怪，为什么段大小的形参为 filesz，而不是类似 segmentsz 之类的？filesz 给人的感觉是文件大小……究其原因是由于程序头中的段大小就叫 p_filesz……好吧不纠结了，反正这也是无伤大雅的事。

将段加载到内存，其实就是我们平时所说的操作系统为用户进程分配内存。程序是由多个段组成的，因此咱们这里按段来处理，分别为每个可加载的段分配内存，内存分配时采用页框粒度。

变量 vaddr_first_page 用于获取虚拟地址 vaddr 所在的页框起始地址。

文件第一个段的起始地址一般情况下都不是自然页，也就是段的起始地址很少有 0xXXXXX000 的情况，多少都会落在页框中的某部分。这种段并未占用完整的自然页，因此要根据段中此部分的尺寸计算出段中其余的尺寸将占用的页框数，将此部分占用的 1 页框与剩余部分占用的页框数加起来才是该段实际需要的页框总数。按照这种思路，变量 size_in_first_page 就表示文件在第一个页框中占用的字节大小，变量 occupy_pages 表示该段占用的总页框数。

第 60 行判断，如果段大小 filesz 大于 size_in_first_page，这表示一个页框容不下该段，下面第 62 行计算该段占用的页框数并赋值给 occupy_pages。第 64 行，如果段比较小，一个页框可以容纳该段，就将 occupy_pages 置为 1。

第 68～84 行“打算”分配 occupy_pages 个页框。为什么说“打算”呢？原因是：exec 是执行新进程，也就是用新进程的进程体替换当前老进程，但依然用的是老进程的那套页表，这就涉及到老进程的页表是否满足新进程内存要求了。如果老进程已经分配了页框，我们不需要再重新分配页框，只需要用新进程的进程体覆盖老进程就行了，只有新进程用到了在老进程中没有的地址时才需要分配新页框给新进程。因此第 71～72 行分别获取新进程虚拟地址 vaddr_page 的 pde 和 pte。第 77 行判断，如果该虚拟地址在老进程中未分配，就调用 get_a_page 分配内存。接着在第 82～83 行更新为下一虚拟页，回到循环开头继续处理。

把段所需要的内存分配好后，下面是从文件系统上加载用户进程到刚刚分配好的内存中，先在第 85 行通过 sys_lseek 函数将文件指针定位到段在文件中的偏移地址，然后第 86 行将该段读入到虚拟地址 vaddr 处。自此，一个段被加载到了内存，见代码 15-21-3。

代码 15-21-3　（project/c15/g/userprog/exec.c）

```
…略
 90 /* 从文件系统上加载用户程序 pathname，
       成功则返回程序的起始地址，否则返回-1 */
 91 static int32_t load(const char* pathname) {
 92     int32_t ret = -1;
 93     struct Elf32_Ehdr elf_header;
 94     struct Elf32_Phdr prog_header;
 95     memset(&elf_header, 0, sizeof(struct Elf32_Ehdr));
 96
 97     int32_t fd = sys_open(pathname, O_RDONLY);
 98     if (fd == -1) {
 99          return -1;
100     }
101
102     if (sys_read(fd, &elf_header, sizeof(struct Elf32_Ehdr)) !=\
        sizeof(struct Elf32_Ehdr)) {
103         ret = -1;
104         goto done;
105     }
106
107     /* 校验 elf 头 */
108     if (memcmp(elf_header.e_ident, "\177ELF\1\1\1", 7) \
109         || elf_header.e_type != 2 \
110         || elf_header.e_machine != 3 \
111         || elf_header.e_version != 1 \
112         || elf_header.e_phnum > 1024 \
113         || elf_header.e_phentsize != sizeof(struct Elf32_Phdr)) {
114         ret = -1;
115         goto done;
116     }
117
118     Elf32_Off prog_header_offset = elf_header.e_phoff;
119     Elf32_Half prog_header_size = elf_header.e_phentsize;
120
121     /* 遍历所有程序头 */
122     uint32_t prog_idx = 0;
123     while (prog_idx < elf_header.e_phnum) {
124        memset(&prog_header, 0, prog_header_size);
125
126        /* 将文件的指针定位到程序头 */
127        sys_lseek(fd, prog_header_offset, SEEK_SET);
128
129        /* 只获取程序头 */
130        if (sys_read(fd, &prog_header, prog_header_size) !=\
             prog_header_size) {
131          ret = -1;
132          goto done;
133        }
134
135        /* 如果是可加载段就调用 segment_load 加载到内存 */
```

```
136         if (PT_LOAD == prog_header.p_type) {
137              if (!segment_load(fd, prog_header.p_offset, \
                    prog_header.p_filesz, prog_header.p_vaddr)) {
138                 ret = -1;
139                 goto done;
140              }
141         }
142
143         /* 更新下一个程序头的偏移 */
144         prog_header_offset += elf_header.e_phentsize;
145         prog_idx++;
146     }
147     ret = elf_header.e_entry;
148     done:
149     sys_close(fd);
150     return ret;
151 }
152
153     /* 用 path 指向的程序替换当前进程 */
154     int32_t sys_execv(const char* path, const char* argv[]) {
155     uint32_t argc = 0;
156     while (argv[argc]) {
157        argc++;
158     }
159     int32_t entry_point = load(path);
160     if (entry_point == -1) {     // 若加载失败，则返回-1
161         return -1;
162     }
163
164     struct task_struct* cur = running_thread();
165     /* 修改进程名 */
166     memcpy(cur->name, path, TASK_NAME_LEN);
167     cur->name[TASK_NAME_LEN-1] = 0;
168
169     struct intr_stack* intr_0_stack = (struct intr_stack*)\
        ((uint32_t)cur + PG_SIZE - sizeof(struct intr_stack));
170     /* 参数传递给用户进程 */
171     intr_0_stack->ebx = (int32_t)argv;
172     intr_0_stack->ecx = argc;
173     intr_0_stack->eip = (void*)entry_point;
174     /* 使新用户进程的栈地址为最高用户空间地址 */
175     intr_0_stack->esp = (void*)0xc0000000;
176
177     /* exec 不同于 fork，为使新进程更快被执行，直接从中断返回 */
178     asm volatile ("movl %0, %%esp; jmp intr_exit" : :\
 "g" (intr_0_stack) : "memory");
179     return 0;
180 }
```

函数 load 接受 1 个参数，可执行文件的绝对路径 pathname，功能是从文件系统上加载用户程序 pathname，成功则返回程序的起始地址，否则返回−1。

函数开头先定义了 elf 头 elf_header 和程序头 prog_header，主要用在第 102 行，读取可执行文件的 elf 头到 elf_header。

第 108 行开始校验 elf 头，判断加载的文件是否是 elf 格式的。

elf 头的 e_ident 字段是 elf 格式的魔数，它是个 16 字节的数组，e_ident[7～15] 暂时未用，因此咱们只需要检测 e_ident[0～6]这七个成员。开头的 4 个字节是固定不变的，它们分别是 0x7f 和字符串“ELF”的 asc 码 0x45、0x4c 和 0x46。成员 e_ident[4]表示 elf 是 32 位，还是 64 位，值为 1 表示 32 位，值为 2 表示 64 位。e_ident[5]表示字节序，值为 1 表示小端字节序，值为 2 表示大端字节序。e_ident[6] 表示 elf 版本信息，默认为 1。以上 e_ident[4-6] 值为 0 均表示非法、不可识别，咱们是在 8086 平台上开发，它是小端字节序，并且是 32 位系统，因此这三位值均取 1，故 e_ident[0-6] 应该分别等于十六进制 0x7F、0x45、0x4C、0x46、0x1、0x1 和 0x1。

第 108 行通过 memcmp 函数比对 elf 头中的 e_ident 魔数，其中“\177”是八进制，十六进制为 0x7f，是 e_ident[0]的固定值。如果您 C 语言开发经验较少，或许您在想“\177”是什么意思，这里多说两句解释下。大伙儿知道，有些不可见字符（多是控制字符）是没法直接通过“键入字符”的方式来表示的，必

须通过其 ASCII 码。在字符串中用 ASCII 码表示字符可以采用“\加 3 位八进制”或“\x 加 2 位十六进制”来表示，注意，字符“\”不可少。回到 elf 魔数，由于字符串“ELF”和 0x7f 前后挨着，并且 E 属于十六进制，故若用“\x7f”表示 0x7f 的话，编译器会把它识别成“\x7fE”LF，也就是 0x7fe，这样就错了，故采用 3 位八进制“\177”来表示。

e_type 表示目标文件类型，其值应该为 ET_EXEC，即等于 2。e_machine 表示体系结构，其值应该为 EM_386，即等于 3。e_version 表示版本信息，其值应该为 1。e_phnum 用来指明程序头表中条目的数量，也就是段的个数，基值应该小于等于 1024。e_phentsize 用来指明程序头表中每个条目的字节大小，也就是每个用来描述段信息的数据结构的字节大小，该结构就是 struct Elf32_Phdr，因此值应该为 sizeof (struct Elf32_Phdr)。因此，第 108～113 行如果不满足任意条件，则认为该文件不是 elf 文件，于是就将返回值 ret 置为−1，跳到标号 done 处，也就是第 149 行，执行“sys_close(fd)”关闭打开的新可执行文件，然后返回 ret 的值。

程序头的起始地址记录在 e_phoff 中，将其获取到变量 prog_header_offset。程序头条目大小记录在 e_phentsize 中，将其获取到变量 prog_header_size 中。下面第 122～146 行，在程序头表中遍历所有程序头。

程序头即段头，段的数量在 e_phnum 中记录，第 123 行 while 循环处理 e_phnum 个段信息。第 127 行通过 sys_lseek 将文件的指针定位到程序头，第 130 行读取段信息到 prog_header 中。第 136 行判断段的类型，如果该段是 PT_LOAD，即可加载的段，那么就调用函数 segment_load 为该段分配内存，从文件系统中加载到内存。循环处理完所有段后，将程序的入口，即 e_entry 作为返回值赋值给 ret，随后关闭可执行文件并返回，至此 load 函数结束。

最后一个函数是 sys_execv，它接受 2 个参数，path 是可执行文件的绝对路径，数组 argv 是传给可执行文件的参数，函数功能是用 path 指向的程序替换当前进程。函数失败则返回−1，如果成功则没机会返回，因此第 176 行的“return 0”只是为满足编译器 gcc 的 c 语法，即“make gcc happy”。

大伙儿已经了解到了，基本上最后要介绍的函数都是前面所有函数的封装，我个人习惯先把所依赖的、基础性的代码放在前面介绍，这种“自下而上”地学习方式深受“好钻研，喜欢抠细节”的同学欢迎。还有一部分同学基础较扎实，细节不用看也能猜得差不多，因此喜欢从上到下，只需要从全局上掌握思路即可。如果您喜欢自上而下先从全局上掌握，可以从后往前看，哈哈，不用我说，我想您已经这样做了，咳咳，看代码。

第 156～158 行通过 while 循环，统计出参数个数，存放到变量 argc 中。

第 159 行调用 load 加载文件 path，成功后，需要修改内核栈中的参数。先在第 164～167 行将 pcb 中的 name 更新为进程名，这样执行 ps 时便会看到正在执行的命令，其中 TASK_NAME_LEN 就是 pcb 中的 name 数组长度，其值为 16，一直纠结于之前的硬编码不太好，所以本次通过宏的方式重新在 thread.h 中定义了。然后在第 169 行获得内核栈的地址，此内核栈中的数据还属于老进程，我们一会要利用该栈从 intr_exit 返回，因此接下来在第 171～175 行修改栈中的数据为新进程。首先将参数数组 argv 的地址赋值给栈中 ebx 寄存器，参数个数 argc 赋值给栈中 ecx 寄存器，新进程从 intr_exit 返回后才是第一次运行，因此运行之初通用寄存器中的值都是无效的，只有运行之后寄存器中的值才是有意义的，故 argc 和 argv 其实放在哪两个通用寄存器中都可以，这里分别将它们放在 ebx 和 ecx 的原因是：ebx 经常做基址寄存器，argv 本来就是所有参数的基地址，ecx 经常做循环控制次数寄存器，argc 本来就是 argv 的参数个数，也就是循环次数，因此属于习惯用法，并不是强制，现在把参数放在哪个寄存器中，将来在获取参数时就从哪些寄存器中取，只要自己协调好就行，将来咱们会实现简易版 c 运行库，那会涉及到从寄存器中获取参数，到时您就会清楚了。接着说代码，第 173 行将可执行文件的入口地址赋值给栈中 eip 寄存器。然后将内核栈中的用户栈指针 esp 恢复为 0xc0000000，这样做的原因有两个，一是老进程用户栈中的数据只适用于老进程，对新进程没用，故新进程的用户栈应该从新开始。二是为了后续传入参数做准备，在很久以前就说过了，用户空间的最高处用于存储命令行参数，以后实现传参时您就清楚了。

接着在第 178 行通过内联汇编，将新进程内核栈地址赋值给 esp 寄存器，然后跳转到 intr_exit，假装从中断返回，实现了新进程的运行。

exec 使程序一去不回头地执行了，因此第 179 行的 return 0 只是为了满足编译器语法要求，其实根本没机会执行。

好啦，sys_execv 就介绍完了，接下来您懂的，添加系统调用 execv，步骤您太熟悉了，咱就不单独贴代码了。

本节到此结束，感谢大伙儿的陪伴。

15.5.2 让 shell 支持外部命令

大伙儿一定听说过，Linux 中执行命令，是 bash（或其他 shell）先 fork 一个子进程，然后调用 exec 去执行命令，其实更严格地说，是执行外部命令时 bash 才会 fork 出子进程并调用 exec 从磁盘上加载外部命令对应的程序，然后执行该程序，从而实现了外部命令的执行，如今我们也效仿这种方式。

由于有了系统调用 exec，我们的 shell 也能调用外部命令了，下面我们改进 shell.c，见代码 15-22。

代码 15-22 （project/c15/g/shell/shell.c）

```
…略
155      } else {      // 如果是外部命令，需要从磁盘上加载
156          int32_t pid = fork();
157          if (pid) {          // 父进程
158      /* 下面这个 while 必须要加上，否则父进程一般情况下会比子进程先执行，
159      因此会进行下一轮循环将 findl_path 清空，
         这样子进程将无法从 final_path 中获得参数*/
160                while(1);
161            } else {          // 子进程
162                make_clear_abs_path(argv[0], final_path);
163                argv[0] = final_path;
164                /* 先判断下文件是否存在 */
165                struct stat file_stat;
166                memset(&file_stat, 0, sizeof(struct stat));
167                if (stat(argv[0], &file_stat) == -1) {
168                      printf("my_shell: cannot access %s:
                         No such file or directory\n", argv[0]);
169                } else {
170                     execv(argv[0], argv);
171                }
172                while(1);
173            }
174        }
175        int32_t arg_idx = 0;
176        while(arg_idx < MAX_ARG_NR) {
177           argv[arg_idx] = NULL;
178           arg_idx++;
179        }
180     }
181     panic("my_shell: should not be here");
182 }
…略
```

从第 155 行起便是对外部命令的处理，当前的进程 shell 先 fork 出子进程，接着在父进程中通过 while(1) 死循环使父进程悬停，即什么都不做。为什么这样安排呢？这是由咱们程序本身的逻辑决定的，并不是什么官方做法。我们在外层循环的开头（可见之前的代码，这部分不属于本次新内容，故未贴出）会清空一些全局数组，如 final_path。父进程一般情况下会比子进程先执行，因此会更快进入下一轮循环将 findl_path 清空，这样子进程将无法从 final_path 中获得参数。

在子进程中，先调用 make_clear_abs_path 获取可执行文件 argv[0]的绝对路径到 final_path 中，然后将 argv[0]重新指向 final_path。接着调用系统调用 stat 判断可执行文件是否存在，如果存在，则执行系统调用 execv 去执行该可执行文件。

参数数组 argv 是由 readline 函数维护的，它会覆盖 argv，并且参数个数是由 argc 来保证的，也不会出现越界的情况。但为了防止出现意外的问题，在第 175～179 行还是清空参数数组 argv，这样更放心一些，调试结束后可以清除掉。

好啦，硬盘上也没有用户程序呢，咱们还真没办法测试，下节咱们再想办法在硬盘上写入用户程序文件，因此本节是欢快的一节，就这么爽快地结束啦。

15.5.3　加载硬盘上的用户程序执行

虽然我很想一次就把用户程序演示完，但担心跨度有点大。那咱们还是按部就班吧，本节先安装个不接受参数的用户程序，也就是用户程序中无 argc 和 argv，没关系，现在最重要的是让用户程序先跑起来。

本节要完成三件事。

（1）编写第一个真正的用户程序。

（2）将用户程序写入文件系统。

（3）在 shell 中执行用户程序，即外部命令。

下面先看用户程序的代码吧，本节新建 command 目录，在其中建立了文件 prog_no_arg.c，如代码 15-23 所示。

代码 15-23　（project/c15/g/command/prog_no_arg.c）

```
1 #include "stdio.h"
2 int main(void) {
3    printf("prog_no_arg from disk\n");
4    while(1);
5    return 0;
6 }
```

代码还是非常简单的，基本上就输出“prog_no_arg from disk”。目前尚未实现系统调用 exit，因此第 4 行的死循环“while(1)”是必须的，没它的话，程序就不知道运行到哪里去了，在此借助这个死循环将程序“卡住”。

下面看编译，具体见代码 15-24。

代码 15-24　（project/c15/g/command/compile.sh）

```
 1 ####  此脚本应该在 command 目录下执行
 2
 3 if [[ ! -d "../lib" || ! -d "../build" ]];then
 4       echo "dependent dir don\`t exist!"
 5       cwd=$(pwd)
 6       cwd=${cwd##*/}
 7       cwd=${cwd%/}
 8       if [[ $cwd != "command" ]];then
 9           echo -e "you\`d better in command dir\n"
10       fi
11       exit
12 fi
13
14 BIN="prog_no_arg"
15 CFLAGS="-Wall -c -fno-builtin -W -Wstrict-prototypes \
16       -Wmissing-prototypes -Wsystem-headers"
17 LIB="../lib/"
18 OBJS="../build/string.o ../build/syscall.o \
19       ../build/stdio.o ../build/assert.o"
20 DD_IN=$BIN
21 DD_OUT="/home/work/my_workspace/bochs/hd60M.img"
22
23 gcc $CFLAGS -I $LIB -o $BIN".o" $BIN".c"
24 ld -e main $BIN".o" $OBJS -o $BIN
25 SEC_CNT=$(ls -l $BIN|awk '{printf("%d", ($5+511)/512)}')
26
27 if [[ -f $BIN ]];then
28       dd if=./$DD_IN of=$DD_OUT bs=512 \
29       count=$SEC_CNT seek=300 conv=notrunc
30 fi
31
32 ##########   以上核心就是下面这三条命令   ##########
33 #gcc -Wall -c -fno-builtin -W -Wstrict-prototypes -Wmissing-prototypes \
34 #   -Wsystem-headers -I ../lib -o prog_no_arg.o prog_no_arg.c
35 #ld -e main prog_no_arg.o ../build/string.o ../build/syscall.o\
36 #   ../build/stdio.o ../build/assert.o -o prog_no_arg
37 #dd if=prog_no_arg of=/home/work/my_workspace/bochs/hd60M.img \
38 #   bs=512 count=10 seek=300 conv=notrunc
```

这是个 shell 脚本，看上去似乎挺复杂，其实核心就是第 33～38 行的三条命令，前面全是配置和判断，没啥含量，而且脚本写得并不严谨，大伙儿不用耐着心思细看脚本了，直接看最后注释的那三条命令即可。这个脚本的功能是自动完成编译、链接、写入硬盘 hd60M.img。脚本最好在 command 目录下执行（其实只要是包括 prog_no_arg.c 的同级目录就行），在脚本前部有判断，执行方式是“sh compile.sh”。

在程序 prog_no_arg.c 中用到了函数 printf，咱们已经加了头文件“stdio.h”，因此在 gcc 编译时添加了“-I ../lib”为其指定头文件目录。由于用到了 stdio.h，故在链接的时候除了要加上 stdio.o 外，还要加上 stdio.h（也是 stdio.o）所依赖的目标文件，它们包括 string.o、syscall.o 和 assert.o。这些目标文件都是 build 目录下的，因此一定要先编译内核，但这并不是说用户程序的库文件依赖于内核的目标文件，只是我们为图省事，复用了这些目标文件，其实单独再写一份库文件也是可以的，不过完全没必要，并不是用了内核的目标文件就是执行了内核的代码，这仅仅表示用户进程中执行的代码和内核目标文件中的代码是一致的，在内存中它们是独立的两份拷贝，互不干涉。总之，无论是用谁的目标文件都不重要，目标（库）文件只是系统调用的封装而已，条条大路都通往北京，不同的库文件最终的出路都是相同的，都是通过系统调用发送 0x80 号中断，利用中断门连接到唯一的内核，殊途同归。提醒一下，一定要注意目标文件的链接顺序，本着“调用在前，定义在后”，否则执行 ld 时可能会提示符号找不到。

也许有同学会有些疑惑：“哎？不是 lib/user 目录下的文件才属于用户进程吗？lib 目录中的文件也可以被随意使用？比如 string.h 经常被内核使用，难道用户进程也可以使用内核代码？”其实代码在哪个目录下这并不重要，目录只是咱们为方便自己开发而规划出来的逻辑结构。任何目录中的代码本质都是普通的文本，并不是说某些文件中的代码只是由内核使用，其他代码由用户进程使用，当然这在逻辑上可以这么划分，但本质上，内核和用户进程只是特权级上的差别，并不是由代码在项目中哪个目录决定的，而是由处理器决定的，也就是代码段寄存器 CS 中选择子的低两位——CPL 位决定的，即处理器的当前特权级。任何代码在处理器眼中都是一样的，差别就体现在执行这些代码时处理器所处的身份——当前特权级 CPL，因此在用户进程中即使包含了内核文件也是没关系的，无非就是特权 3 级下去执行同内核一样的指令，除非遇到某些只能在特权级 0 下执行的指令时才会报错，如 pushf、sti 等，对于那么特权级不敏感的普通指令，处理器一律执行。换句话说，处理器根本不知道当前执行的指令属于内核，还是用户进程，而且说实在的，它也没办法直接知道，甚至连内核级和用户级都不知道是什么，因为这是人来界定的概念嘛，处理器只知道 0、1、2、3 这四个特权级。人们习惯让内核在 0 特权级下运行，因此可以把 0 特权级称为内核级，但谁说一定得这样了，处理器提供了四个特权级呢，因此特权级 1 也可以做所谓的内核级，特权级 2 可以做用户级，剩下的两个特权级 0 和 3 就是空着不用也行的，只是大多数人不这么做而已。也许有同学会说，可以通过指令的地址来判断，如果指令地址在 0xc0000000 以上，这就说明是内核的数据或指令。没错，编译器会根据咱们的需求把内核代码的地址编译为 0xc0000000 以上，但这只是咱们人为安排的，并不是处理器硬件一级的要求。咱们高兴的话，完全可以把 32 位的 4GB 空间平分给内核和用户空间，如 0x80000000 以上的空间是内核。总之地址空间划分是为人设计的管理策略，和处理器无关，并不能作为判断内核指令或用户指令的依据。处理器在任何特权级下都可以执行特权级不敏感的指令，因此原则上只要库文件中不包含特权级敏感指令，用户进程可以包括内核的库文件，内核也可以包含用户进程的库文件。

下面看如何把程序写入文件系统，也就是写入 hd80M.img 中。

这里说的是把程序写入“文件系统”，不是写入“硬盘”，这还是有点区别的。写入硬盘和写入文件系统虽说最终都是往硬盘上写入数据，但方式是不一样的。写入硬盘完全可以直接用 dd 命令或者硬盘驱动直接“生硬地”往某个扇区填数据，而写入文件系统则是把数据按照文件系统的规则写入硬盘，这涉及到文件系统元信息的同步维护，否则就会破坏文件系统。shell 通过文件系统来获取外部命令，我们只在 hd80M.img 上创建了文件系统，因此程序必须写入到 hd80M.img 中，并且把程序写入硬盘的操作必须要通过文件系统函数才行，不能绕过它们强行写入。

这是由两步来完成的，为了不破坏 hd80M.img 上的文件系统，第 1 步先将文件写入到 hd60M.img 中，这是由脚本中最后一个命令 dd 完成的，由它负责把编译出来的二进制文件 prog_no_arg 写入到硬盘 hd60M.img。hd60M.img 是裸盘，无文件系统，因此可以随便写入而不存在破坏文件系统的问题。本例中

把它写到了 hd60M.img 中偏移 300 个扇区的位置，即“seek=300”，写入了 10 个扇区大小的数据，即“count=10”。偏移 300 扇区的原因是咱们的内核文件也是在 hd60M.img 中，目前其大小约为 73KB 左右，大概占 142 个扇区。程序文件 prog_no_arg 也是写在 hd60M.img 中，因此偏移 300 扇区足够跨过内核程序，避免破坏内核。为什么写入 10 个扇区的数据呢？原因是程序 prog_no_arg 大小为 4777 字节（脚本中会自动判断文件大小），它是通过 ls–l 命令获得的，ls 命令的输出信息中包括文件的大小，这里就不单独贴图了，大伙儿只要知道 prog_no_arg 大小为 4777 字节就行了。4777 字节至少占用了 10 个扇区。脚本 compile.sh 写得也不严谨，仅是为了编译时方便，不细说了，您只要了解核心三条命令的意义即可。有兴趣查看脚本执行细节的同学，您可以在 command 目录下执行“sh –x compile.sh”，shell 会以调试方式执行并输出信息。

第 2 步是将 hd60M.img 上的程序读出来，再通过文件系统函数写入 hd80M.img 中。这得写代码来完成，具体是在 main.c 中加入了读取 prog_no_arg 的代码，见代码 15-25。

代码 15-25　（project/c15/g/kernel/main.c）

```
…略
21 int main(void) {
22      put_str("I am kernel\n");
23      init_all();
24
25 /*************    写入应用程序    *************/
26      uint32_t file_size = 4777;
27      uint32_t sec_cnt = DIV_ROUND_UP(file_size, 512);
28      struct disk* sda = &channels[0].devices[0];
29      void* prog_buf = sys_malloc(file_size);
30      ide_read(sda, 300, prog_buf, sec_cnt);
31      int32_t fd = sys_open("/prog_no_arg", O_CREAT|O_RDWR);
32      if (fd != -1) {
33           if(sys_write(fd, prog_buf, file_size) == -1) {
34               printk("file write error!\n");
35               while(1);
36           }
37      }
38 /*************    写入应用程序结束    *************/
39      cls_screen();
40      console_put_str("[rabbit@localhost /]$ ");
41      while(1);
42      return 0;
43 }
…略
```

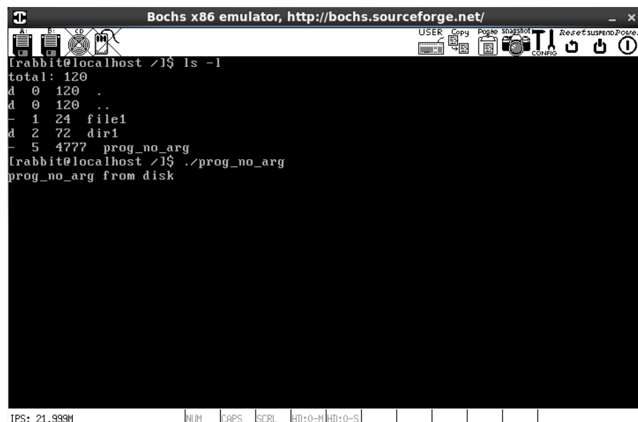
第 26～37 行的代码是本次写入文件的代码，在文件写入之后就没用了，下次再运行时记得注释掉。

第 26 行的 file_size 是 prog_no_arg 的字节大小，其值正如前所说是 4777 字节。下一行的 sec_cnt 是 prog_no_arg 占用的扇区数。第 28 行指定操作的设备是 sda，即第 0 个 ide 通道上的第 0 块硬盘。

第 29～31 行，以 sba 上第 300 个扇区为起始，读取 sec_cnt 个扇区到缓冲区 prog_buf。

第 31～33 行在根目录下创建文件 prog_no_arg，并将缓冲区中的数据，也就是程序文件写入 prog_no_arg。

下面就差第三件事没做了，在 shell 中执行 prog_no_arg 测试，结果如图 15-12 所示。



▲图 15-12　无参数的用户进程

图中先执行了“ls –l”命令，列出了根目录下的所有文件。其中包括 prog_no_arg，已经写入成功。下面执行“./prog_no_arg”执行该程序，输出了“prog_no_arg from disk”。目测符合预期，这说明我们第一个真正的用户进程成功了，我去喝瓶酸奶庆祝下。

本节就到这了，下节咱们想办法让用户进程接受参数，兄弟们下节见。

15.5.4 使用户进程支持参数

我们为内部命令传递参数还是很简单的，在调用内部命令时直接代入参数就行了，但对于外部命令，参数传递就没那么简单了。

大伙儿知道，根据 C 调用约定，参数是通过栈来传递的，在同一个进程中，由于栈已经存在了，参数可以直接压在栈中，被调函数便可以从栈中获取参数了，这使得咱们处理内部命令时很容易。获取参数是在执行命令之前，要想把获取到的参数传递给某个命令，该命令所属的进程必须先有栈。但外部命令的执行，实质上是加载一个用户程序的过程，shell 执行外部命令前，外部命令（用户进程）尚未创建，它的栈当然也不存在了，参数就没法传递了吗？这该怎么办呢？

您看，我们在编写用户程序时都要有个 main 函数，它是主函数，可能在初次接触编程时便被教导：main 函数是第 1 个函数。其实并不是这样，main 函数的声明一般是“void main(argc, argv)”、“int main(argc, argv)”、“void main()”等，通过这几种声明，我们至少可以看出两件事。

（1）无论返回值和形参如何变动，不变的是函数名 main，在标准情况下它永远为 main（当然它也可以变，一会您就知道了）。

（2）main 函数有形参，参数是被调用时才传递的，这说明 main 函数也是被别人调用后才运行的，因此 main 函数其实并不是真正的第 1 个函数。main 函数数声明中有返回值类型，这充分说明 main 是被调用才执行的。

想想看，既然 main 是被调用执行的，并且有返回值，那么它执行完成后，程序会返回到哪里？或者说，为什么 main 函数要返回？

原来，为了让用户可以更方便地编程，前辈们开发了大量的标准化框架，也就是各种库，其中很著名的就是 C 标准库和 C 运行库。

为什么要有标准库？这个是历史发展必然的结果。任何程序员为了方便自己开发程序，都会把一些常用的功能写成通用的函数，以后随时调用，使开发效率提升。随着平时积累的通用函数越来越多，渐渐形成规模，也就是形成了早期的函数库，属于此库中的函数就称为库函数。当然这个库中通用的函数有可能是多个人联合开发的，比如项目组中每个人负责开发一块，这样每个人都积累出一些通用函数，大伙儿一合计，把每个人的通用代码集成到一起咱们一块用吧，这也是函数库。与此同时，所有的项目组都在开发自己的函数库。但很明显，这种开发出来的函数库具有很强的本地性，即某个项目组开发出来的库只被本项目组的人接受，并不通用，因为很有可能其他项目组的人会想，我们组也实现了一个叫“×××”的函数，参数比你们的少，但功能更为强大，你们做的还不如我们的好，应该用我们的库。这还只是项目组之间的矛盾，在项目组内部也同样出现类似的情形，某个成员认为同组人开发出来的函数，在名称上不合适或者在功能上有所欠缺，很不情愿地被动接受。类似重复造轮子的情况比比皆是，各个单位、公司，甚至教育机构都开发出自己的库，没有统一的标准接口，大家都认为自己的好，总之谁也不服谁，这时候需要有个大家都服的老大出来震慑，它就是美国国家标准协会，即 ANSI（American National Standards Institute），它规定出一套 C 函数标准接口，也就是今天所说的 C 标准库，明确规定了每个函数的作用及原型，大家要共同遵守。

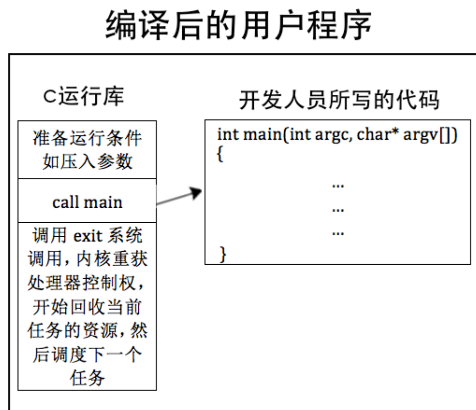
C 标准库是与操作系统平台无关的，它诞生之初就是为了实现用户程序跨操作系统平台而规约的标准接口，使用户进程无论在哪个操作系统上调用同样的函数接口，执行的结果都是一样的。

C 运行库也称为 CRT（C RunTime library），它是与操作系统息息相关的，因为谁也不愿意重复造轮子，故它的实现也基于 C 标准库，因此 CRT 属于 C 标准库的扩展。CRT 多是补充 C 标准库中没有的功能，为适配本操作系统环境而定制开发的。因此 CRT 并不通用，只适用于在本操作系统上运行的程序。

CRT 都做了什么呢？很多，最主要的就是初始化运行环境，在进入 main 函数之前为用户进程准备条

件，传递参数等，待条件准备好后再调用用户进程的 main 函数，这样用户进程才能顺利地跑起来。当用户进程结束时，CRT 还要负责回收用户进程的资源。其实想想这也是必然的，main 函数是用户自己写的，无论代码多少，总有结束那天（死循环不算），如果 main 执行到了边界，此时没有固定的代码执行，程序不就“飞”了吗，也就是说处理器会越过边界自动向下取指令，cs:eip 寄存器中的值肯定就不对了，因此程序不知道会跑哪里去了，处理器一直会执行到抛异常为止，操作系统也就失去了处理器的控制权，整个计算机系统瘫痪了，这就是咱们经常在程序的最后添加死循环代码“while(1)”的原因。综上所述，main 函数一定是被 call 指令调用的，必须有去有回，目的是当用户进程执行完用户所写的 main 函数后能够执行固定的代码——系统调用 exit 或 _exit，这样用户程序陷入内核，使处理器的控制权重新回到操作系统手中。

其实 CRT 代码才是用户程序的第一部分，我们的 main 函数实质上是被夹在 CRT 中执行的，它只是用户程序的中间部分，编译后的二进制可执行程序中还包括了 CRT 的指令，其结构如图 15-13 所示。



▲图 15-13　CRT 与用户程序

您看，咱们似乎要做个 CRT 了，其实我这么说的时候也吓了我自己一跳，哈哈，我可没那么大的本事，咱们就写个功能类似的简易版本吧，见代码 15-26。

代码 15-26　（project/c15/h/command/start.S）

```
1 [bits 32]
2 extern  main
3 section .text
4 global _start
5 _start:
6     ;下面这两个要和 execv 中 load 之后指定的寄存器一致
7     push  ebx     ;压入 argv
8     push  ecx     ;压入 argc
9     call  main
```

这一次我们是在 command 目录中创建 start.S，它是用户程序真正的第一个函数，是程序的真正入口，这是我们编译后的用户程序中的第一部分。代码贴出来后我都觉得有些惭愧了，代码何止是简易，简直是简陋，虽然是汇编代码，但真的做到了零难度，不过对目前来说已经够用了，以后还会稍微增加点代码。

第 2 行通过“extern main”声明了外部函数 main，这个就是咱们用户程序中的主函数 main。介绍到这顺便说一下，main 函数名其实也可以用其他名称来替换，无论是什么名字作为主函数，这里要用 extern 声明它。咱们就不单独测试了，很简单的，大伙儿感兴趣自行测试吧。

第 5 行是标号_start，它是链接器默认的入口符号，如果 ld 命令链接时未使用链接脚本或-e 参数指定入口符号的话，默认会以符号_start 为程序入口。我们这里就用这个默认的_start。

在文件 exec.c 中我们已经把新进程的参数压入内核栈中相应的寄存器，sys_execv 执行完成从 intr_exit 返回后，寄存器 ebx 是参数数组 argv 的地址，寄存器 ecx 是参数个数 argc。因此在第 7～8 行将它们压入栈，此时的栈是用户栈，这就是之前咱们所说的，自己和自己协调好就行了：在 sys_execv 中，往 0 特权级栈中哪个寄存器写入参数，此处就从哪个寄存器中获取参数，然后再压入用户栈为用户进程准备参数。

第 9 行通过 call 指令调用外部函数 main，也就是用户程序开发人员所负责的主函数 main，至此，用户程序的主函数开始运行。

好啦，参数传递的问题解决了，下面看下本次的用户程序吧，见代码 15-27。

代码 15-27　（project/c15/h/command/prog_arg.c）

```
1 #include "stdio.h"
2 #include "syscall.h"
3 #include "string.h"
```

```
 4 int main(int argc, char** argv) {
 5     int arg_idx = 0;
 6     while(arg_idx < argc) {
 7        printf("argv[%d] is %s\n", arg_idx, argv[arg_idx]);
 8        arg_idx++;
 9     }
10     int pid = fork();
11     if (pid) {
12         int delay = 900000;
13         while(delay--);
14         printf("\n     I`m father prog, my pid:%d,
           I will show process list\n", getpid());
15         ps();
16     } else {
17         char abs_path[512] = {0};
18         printf("\n     I`m child prog, my pid:%d,
           I will exec %s right now\n", getpid(), argv[1]);
19        if (argv[1][0] != '/') {
20            getcwd(abs_path, 512);
21            strcat(abs_path, "/");
22            strcat(abs_path, argv[1]);
23            execv(abs_path, argv);
24        } else {
25            execv(argv[1], argv);
26        }
27     }
28     while(1);
29     return 0;
30 }
```

大体上说下代码，这次的测试内容丰富一些，先在函数开头打印了接受的参数，参数 argv[0]是本程序的名称，即 prog_arg，参数 argv[1]是让 prog_arg 去执行的可执行文件的路径。

第 10 行调用 fork 系统调用派生出子进程。父进程先用 while 循环了 900000 次 delay--，相当于一段时间延迟，目的是避免和子进程输出信息混杂在一起。然后调用 getpid 系统调用打印出自己的 pid，最后调用 ps 系统调用打印任务信息，此时在任务列表中应该包括子进程的信息了。在子进程中先打印自己的 pid 及输出信息，表示自己马上要执行程序 prog_no_arg，也就是参数 argv[1]中的可执行文件。第 19 行判断 argv[1]是相对路径，还是绝对路径，如果是相对路径，调用 getcwd 系统调用获得工作目录，然后将 argv[1]追加到工作目录之后，最后在第 23 行调用 execv 去执行它。如果 argv[1]是绝对路径，直接在第 25 行调用 execv 执行即可。

代码写得很不严谨，意思到了就行，请大伙儿包涵。下面看编译脚本，见代码 15-28。

代码 15-28　（project/c15/h/command/compile.sh）

```
 1 ####  此脚本应该在 command 目录下执行
 2
 3 if [[ ! -d "../lib" || ! -d "../build" ]];then
 4       echo "dependent dir don\`t exist!"
 5       cwd=$(pwd)
 6       cwd=${cwd##*/}
 7       cwd=${cwd%/}
 8       if [[ $cwd != "command" ]];then
 9             echo -e "you\`d better in command dir\n"
10       fi
11       exit
12 fi
13
14 BIN="prog_arg"
15 CFLAGS="-Wall -c -fno-builtin -W -Wstrict-prototypes \
16             -Wmissing-prototypes -Wsystem-headers"
17 LIBS="-I ../lib -I ../lib/user -I ../fs"
18 OBJS="../build/string.o ../build/syscall.o \
19             ../build/stdio.o ../build/assert.o start.o"
20 DD_IN=$BIN
21 DD_OUT="/home/work/my_workspace/bochs/hd60M.img"
22
23 nasm -f elf ./start.S -o ./start.o
24 ar rcs simple_crt.a $OBJS start.o
25 gcc $CFLAGS $LIBS -o $BIN".o" $BIN".c"
```

```
26 ld $BIN".o" simple_crt.a -o $BIN
27 SEC_CNT=$(ls -l $BIN|awk '{printf("%d", ($5+511)/512)}')
28
29 if [[ -f $BIN ]];then
30        dd if=./$DD_IN of=$DD_OUT bs=512 \
31        count=$SEC_CNT seek=300 conv=notrunc
32 fi
33
34 ##########   以上核心就是下面这五条命令   ##########
35 #nasm -f elf ./start.S -o ./start.o
36 #ar rcs simple_crt.a ../build/string.o ../build/syscall.o \
37 #   ../build/stdio.o ../build/assert.o ./start.o
38 #gcc -Wall -c -fno-builtin -W -Wstrict-prototypes -Wmissing-prototypes \
39 #   -Wsystem-headers -I ../lib/ -I ../lib/user -I ../fs prog_arg.c -o prog_arg.o
40 #ld prog_arg.o simple_crt.a -o prog_arg
41 #dd if=prog_arg of=/home/work/my_workspace/bochs/hd60M.img \
42 #   bs=512 count=11 seek=300 conv=notrunc
```

脚本只是为了方便扩展和快速编译，大伙儿还是直接看最下面的五条核心命令吧。和上节的compile.sh类似，不过是多了两个命令，先用nasm把start.S编译为start.o。为了方便，这里还调用了ar命令，将string.o、syscall.o、stdio.o、assert.o和start.o打包成静态库文件simple_crt.a，simple_crt.a类似于CRT的作用，它就是我们所说的简陋版C运行库。后面的用户程序目标文件prog_arg.o和它直接链接就可以了。后面的指令之前说过了，下面看main.c中的代码，见代码15-29。

代码 15-29　（project/c15/h/kernel/main.c）

```
…略
21 int main(void) {
22      put_str("I am kernel\n");
23      init_all();
24
25 /*************    写入应用程序    *************/
26      uint32_t file_size = 5307;
27      uint32_t sec_cnt = DIV_ROUND_UP(file_size, 512);
28      struct disk* sda = &channels[0].devices[0];
29      void* prog_buf = sys_malloc(file_size);
30      ide_read(sda, 300, prog_buf, sec_cnt);
31      int32_t fd = sys_open("/prog_arg", O_CREAT|O_RDWR);
32      if (fd != -1) {
33           if(sys_write(fd, prog_buf, file_size) == -1) {
34               printk("file write error!\n");
35               while(1);
36           }
37      }
38 /*************    写入应用程序结束   *************/
39      cls_screen();
40      console_put_str("[rabbit@localhost /]$ ");
41      while(1);
42      return 0;
43 }
…略
```

依然还是老样子，只是在第31行往根目录中写入的文件名是prog_arg。好啦，编译运行看结果，如图15-14所示。

图中先执行了“ls –l”命令查看了文件的写入情况，果然prog_arg已经写入到根目录下，然后执行“./prog_arg /prog_no_arg”，意图是让prog_arg去调用我们上节中完成的用户程序prog_no_arg。执行过后，屏幕上输出了参数名，然后父进程执行90万次的空循环，随后子进程被调度，打印出“I'm child prog, my pid:6…”，然后执行程序prog_no_arg，prog_no_arg执行后输出“prog_no_arg from disk”。父进程的空循环执行过后，姗姗来迟地输出了“I'm father prog, my pid:5…”。接着打印了任务信息，其中pid为5的任务是父进程，其STAT为RUNNING，其执行的COMMAND是/prog_arg。pid为6的是子进程，其STAT为READY，其执行的COMMAND是/prog_no_arg。

目测运行符合预期，本节到此结束，下节再见。

```
Bochs x86 emulator, http://bochs.sourceforge.net/
[rabbit@localhost /]$ ./prog_arg /prog_no_arg
argv[0] is /prog_arg
argv[1] is /prog_no_arg

      I`m child prog, my pid:6, I will exec /prog_no_arg right now
prog_no_arg from disk

      I`m father prog, my pid:5, I will show process list
PID             PPID            STAT            TICKS           COMMAND
1               NULL            READY           7D8A0           init
2               NULL            READY           7D8C0           main
3               NULL            BLOCKED         0               idle
4               1               READY           2820            init
5               4               RUNNING         2728            /prog_arg
6               5               READY           26C0            /prog_no_arg

IPS: 28.598M      NUM  CAPS  SCRL  HD:0-M  HD:0-S
```

▲图 15-14　接受参数的用户进程

15.6　实现系统调用 wait 和 exit

大伙儿可能经常会使用 exit，它是如此普遍，以至于无论是在哪种语言中都会有它的影子。它还有一个好兄弟——wait，它其实总是与 exit 配合使用，这两兄弟虽然是成对使用的，但未必都常用，尤其是 wait，初次接触时不容易理解其作用，甚至对开发经验较少的同学来说，wait 是“最熟悉的陌生人”。

15.6.1　wait 和 exit 的作用

由于 wait 和 exit 是成对使用的好兄弟，咱们就不把它们拆开了，一块说吧。

无论是业务上的需要，还是调试需要，大多数同学实际工作中经常使用 exit、_exit 或其他功能类似的系统调用，exit 的作用很直白，就是使进程“主动”退出，结束运行。其实在图 15-13 中已经透露了一件事，在 C 运行库中调用 main 函数执行，main 函数执行结束后程序流程会回到 C 运行库，C 运行库的结束代码处会调用 exit。这表明任何时候进程都会调用 exit，即使程序员未写入调用 exit 的代码，在 C 运行库的最后也会发起 exit 的调用。由此可见，结束程序运行始终是通过主动调用 exit 系统调用实现的，因为这是唯一让系统重新拿回处理器控制权的机会。

可能有些同学对 wait 有些不解，不知道其具体是干吗的。wait 的作用是阻塞父进程自己，直到任意一个子进程结束运行。wait 通常是由父进程调用的，或者说，尽管某个进程没有子进程，但只要它调用了 wait 系统调用，该进程就被认为是父进程，内核就要去查找它的子进程，由于它没有子进程，此时 wait 会返回-1，表示其没有子进程。如果有子进程，这时候该进程就被阻塞，不再运行，内核就要去遍历其所有的子进程，查找哪个子进程退出了，并将子进程退出时的返回值传递给父进程，随后将父进程唤醒。

读了上面这段文字似乎对 wait 明白了一些，但似乎又没说到点上。上面这段文件解释了 wait 的作用主要有两个，一是使父进程阻塞，二是获得子进程的返回值。其实第二个作用和第一个作用相比“显得”没那么重要，wait 的主要作用就是使父进程阻塞。大伙儿想想看，什么是阻塞？阻塞是指任务不在就绪队列当中，这样调度器就不会调度它，因此该任务就不会运行。这里的重点就是“不运行”，这才是 wait 的使命，通过阻塞父进程，可以解决父子进程同步的问题，其实咱们已经遇到了这种问题了。

上一节中，为了使父子进程输出的信息不混杂在一起，咱们特意在父进程中通过 while 执行了个 90 万次的无意义的循环以实现延迟。先不说浪费宝贵的 CPU 资源，这种空兜 CPU 的方式也并不是万全之策，那个 while

循环产生的延迟取决于实际 CPU 的主频，如果 CPU 主频很低，自然延迟的时间相对较长，如果主频比较高，10 万次的循环也起不到多大的作用，父子进程的信息还是会混杂在一起。除此之外，进程的调度时机是由调度器的调度算法决定的，如果系统很忙，父进程把 10 万次循环都执行完了，调度器却一直未开始新的调度，当父进程又打印了一句话后，此时调度器开始调度子进程上 CPU，子进程开始输出，这下父子进程的信息又混杂在一起了，您看，还是没从根本上解决问题，总之空兜 CPU 实现的延迟只是在大多数情况下有效。想想看，其实这本质上属于进程同步的问题，也就是协调父子进程某些代码的执行次序，我们希望子进程执行完某些代码后再让父进程执行。对于父子进程的同步，可以用 wait 系统调用来解决，这其实就是 wait 的初心。

也许我们对 wait 的概念有些模糊的原因是它不像 exit 那样表意直白，从名字上就能理解其功能，wait 是等待的意思，初次接触时不禁要问了，等待什么？如果 wait 不叫 wait，而是更直白地叫作诸如“block_myself”，甚至更直白一点：“let_child_execute_first”，我想大伙儿就不会对其作用感到拿捏不准了。

好啦，本节结束啦。

15.6.2　孤儿进程和僵尸进程

话说 Linux 系统中为什么有孤儿进程和僵尸进程？原因是因为有 wait 和 exit 系统调用。

前面介绍到，进程到最后都会调用 exit 结束运行，无论它是主动调用的 exit，还是 C 运行库中的 exit。“如果一个子进程的运行结束了，它的父进程没有调用 wait（Linux 中还有 waitpid 也是同样的功能），那么该进程就变成僵尸进程。”这是大多数对僵尸进程的解释，我当初接触这个概念时不禁发问，为什么父进程不 wait 的话，子进程就变成僵尸？或者说，父进程到底在“等”什么？父进程想从子进程那里等来什么？什么东西这么重要以至于子进程不交付给父进程的话会变成僵尸而死不瞑目？

前面咱们也说过了，wait 的一个作用就是阻塞父进程，使父子进程同步，另外一个作用就是获得子进程的“退出状态”。父进程派生出子进程的目的是让子进程帮忙做一些工作，子进程在其有限的生命中要拼命工作，但未必会把工作成功完成，这就像咱们的实际工作一样，有的工作难度较低，很容易完成，有的工作难度较大，最终失败了，子进程工作完成的成功与否，不能光子进程自己知道，还得向上级汇报，子进程是由父进程委派的，因此必须要告诉父进程自己的任务完成了没有。怎样汇报呢？这是通过子进程的返回值体现的，也就是子进程 main 函数中最后的 return 语句的值，就是进程所谓的“退出状态”。当子进程执行完 main 函数后，程序流程会回到 C 运行库，C 运行库会把进程 return 的返回值通过系统调用 exit 提交给内核。这是子进程的主函数全部执行完的情况，如果进程还没到 return 就想半路主动退出呢？其返回值该如何传递给父进程？这个好办，因为 exit 的原型就是“void _exit(int status)”，其中 status 就是子进程的返回值，C 运行库中调用 exit 的形式就是 exit(子进程的返回值)，那子进程直接调用 exit(返回值)就可以了，这就是咱们调用 exit 时必须要提供个返回值的原因。其实子进程的返回值并不是手递手直接交给父进程的，您想，进程都是独立的地址空间，即使是父子进程，它们之间也是相互独立不可互访的，因为这就是与线程的区别，进程间要想相互通信必须要借用内核（无论是管道、消息队列，还是共享内存等进程间通信形式，无一例外都是借助内核这个中间人），子进程的返回值肯定是先要交给内核，然后是父进程向内核要子进程的返回值。在子进程的返回值提交给内核后，父进程该如何向内核要子进程的返回值呢？这时系统调用 wait 的第二个作用就发挥出来了，wait 的原型是“pid_t wait(int *status)”，其中 status 是父进程用于存储子进程返回值的地址，父进程调用它之后，内核就会把子进程的返回值存储到 status 指向的内存空间，至此父进程终于了解了子进程的临终遗言，也就是退出状态。

大伙儿有没有想过，进程是单独执行的个体，每个进程都有自己的返回值，那返回值存放在哪里呢？估计您也想到了，为了方便管理，与进程相关的数据都统一放在 pcb 中，当进程生命结束时，它的遗言，也就是返回值，会被内核放在 pcb 中。另外，进程在调用 exit 时就表示进程生命周期结束了，其占用的资源可以被回收了，因此进程在调用 exit 后，内核会把该进程占用的大部分资源都回收，比如内存、页表等，但肯定不能将进程的 pcb 所占的内存回收，原因是里面存储着子进程的遗言，必须要交付给父进程，父进程收到子进程的遗言后才能回收子进程的 pcb，否则子进程会“死不瞑目”。这表示进程的 pcb 是进程最后占用的资

源，它应该在父进程调用 wait 获取子进程的返回值后，再由内核回收子进程 pcb 所占的 1 页框内存，也就是说，回收 pcb 内存空间的工作是在系统调用 wait 对应的内核实现中。子进程 pcb 的回收是发生在父进程调用 wait 之后、内核受理系统调用的期间，因此"相当于"父进程为子进程"收尸"，其实是内核为子进程"收尸"。

小总结一下，进程结束时会通过 exit 留下点"遗言"，也就是返回值，它代表了子进程这一生工作的结果，父进程为了获知子进程的成果如何，必须要获得子进程的返回值，而获得子进程返回值的方法，就是父进程调用 wait 系统调用。如果把父子进程之间的通信比喻成邮信，子进程通过 exit 来给父进程写信，exit 把信交给了内核，父进程知道子进程一定会写信给它，因此它主动调用 wait 收信，内核在父子进程之间起到了邮递员的作用，把子进程的返回值投递给父进程。

以上是正常情况下的父子进程间的通信，说了这么多，下面该解释孤儿进程和僵尸进程了。

在子进程提交给父进程返回值的通信中，有这样一种情况，当父进程提前退出时，它所有的子进程还在运行，没有一个执行了 exit，因为它们的生命周期尚未结束，还在运行中，个个都拥有"全尸"（进程体），这些进程就称为孤儿进程。这时候所有的子进程会被 init 进程收养，init 进程会成为这些子进程的新父亲，当子进程退出时会由 init 负责为其"收尸"。其实想想这也是顺理成章的，毕竟 init 进程是所有进程的父进程，将子进程托付给 init 是再合理不过的。

僵尸进程是怎么回事呢，僵尸进程也称为 zombie，下面还拿子进程提交返回值来举例说明。如果父进程在派生出子进程后并没有调用 wait 等待接收子进程的返回值，这时某个子进程调用 exit 退出了，自然没人来接收返回值了（父进程未退出，因此子进程不能过继给 init，init 也不能帮子进程做善后收尸，只有父进程才有权限为子进程收尸），因此其 pcb 所占的空间不能释放，没人为其"收尸"，自然就成了"僵尸"。说白了，僵尸进程就是针对子进程的返回值是否成功提交给父进程而提出的，父进程不调用 wait，就无法获知子进程的返回值，从而内核就无法回收子进程 pcb 所占的空间，因此就会在队列中占据一个进程表项。因此您懂的，僵尸进程是没有进程体的，因为其进程体已在调用 exit 时被内核回收了，现在只剩下一个 pcb 还在进程队列中，它并不占太多的资源。在 Linux 中，用 ps 命令查看的任务列表当中，stat 为"Z"的进程就是僵尸进程，也就是 Zombie。

对系统而言，有了 init 进程的"收养"，孤儿进程并没有什么危害， init 会很好地为其善后，因此并不会额外占用资源，它和普通的进程一样，原理上对系统不会产生不良影响。

下面多说两句僵尸进程，僵尸进程的本质是不占资源，仅含有进程表项 pcb，理由如下。

首先进程退出状态得保存在某处，保存在 pcb 中，这是最合适的选择，至少节省了单独保存退出状态的空间，并且由于每个进程都有唯一的退出状态，放在 pcb 中容易与进程相关联，好管理。

其次，进程的退出状态未被父进程取出前，除了 pcb 以外，其他所有资源都可以释放。

以上是僵尸进程 pcb 必然残留的原因，部分程度上解释了僵尸进程的合理性，似乎没什么危害，是这样吗？咱们继续看。僵尸进程虽然没有进程体，只在内存中保留一个 pcb，但由于 pcb 不释放，它原本的 pid 也会继续被占用，当僵尸进程数量很大时，系统将无可用 pid 分配给新进程，从而加载进程失败。然后僵尸进程并不是问题所在，问题的根源在于产生僵尸进程的父进程，因此对于这种情况，就要将僵尸进程的父进程 kill 掉。在 Linux 中可以利用 ps –ef 查看所有任务的 pid 和 ppid，找到状态为 Z 的进程，查看其 ppid，跟着向 pid 为 ppid 的进程发送 kill -9，手起刀落，系统又和谐了。

总结：

exit 是由子进程调用的，表面上功能是使子进程结束运行并传递返回值给内核，本质上是内核在幕后会将进程除 pcb 以外的所有资源都回收。wait 是父进程调用的，表面上功能是使父进程阻塞自己，直到子进程调用 exit 结束运行，然后获得子进程的返回值，本质上是内核在幕后将子进程的返回值传递给父进程并会唤醒父进程，然后将子进程的 pcb 回收。

好啦，本节终于结束了，基础内容就介绍到这，大伙儿下节见。

15.6.3 一些基础代码

在进一步实现 exit 和 wait 之前，有一些基础工作还是必不可少的，下面分别介绍下。

进程的返回值，也就是退出状态存储在 pcb 中，因此我们要在 pcb 中增加一个成员来记录返回值，现在完整的 pcb 如代码 15-30 所示。

代码 15-30　（project/c15/i/thread/thread.h）

```
…略
 78 /* 进程或线程的 pcb，程序控制块 */
 79 struct task_struct {
 80     uint32_t* self_kstack;   // 各内核线程都用自己的内核栈
 81     pid_t pid;
 82     enum task_status status;
 83     char name[TASK_NAME_LEN];
 84     uint8_t priority;
 85     uint8_t ticks;          // 每次在处理器上执行的时间嘀嗒数
 86 /* 此任务自上 cpu 运行后至今占用了多少 cpu 嘀嗒数，
 87  * 也就是此任务执行了多久*/
 88     uint32_t elapsed_ticks;
 89 /* general_tag 的作用是用于线程在一般的队列中的结点 */
 90     struct list_elem general_tag;
 91 /* all_list_tag 的作用是用于线程队列 thread_all_list 中的结点 */
 92     struct list_elem all_list_tag;
 93     uint32_t* pgdir;             // 进程自己页表的虚拟地址
 94     struct virtual_addr userprog_vaddr;   // 用户进程的虚拟地址
 95     struct mem_block_desc u_block_desc[DESC_CNT];
                                 // 用户进程内存块描述符
 96     int32_t fd_table[MAX_FILES_OPEN_PER_PROC];   // 已打开文件数组
 97     uint32_t cwd_inode_nr;        // 进程所在的工作目录的 inode 编号
 98     pid_t parent_pid;          // 父进程 pid
 99     int8_t  exit_status;         // 进程结束时自己调用 exit 传入的参数
100     uint32_t stack_magic;
                         // 用这串数字做栈的边界标记，用于检测栈的溢出
101 };
…略
```

其中第 99 行的 exit_status 就是进程的退出状态值。

除此之外还要添加个内存释放的函数，这定义在 memory.c 中，见代码 15-31。

代码 15-31　（project/c15/i/kernel/memory.c）

```
…略
578/* 根据物理页框地址 pg_phy_addr 在相应的内存池的位图清 0，不改动页表*/
579 void free_a_phy_page(uint32_t pg_phy_addr) {
580     struct pool* mem_pool;
581     uint32_t bit_idx = 0;
582     if (pg_phy_addr >= user_pool.phy_addr_start) {
583          mem_pool = &user_pool;
584          bit_idx = (pg_phy_addr - user_pool.phy_addr_start) / PG_SIZE;
585     } else {
586          mem_pool = &kernel_pool;
587          bit_idx = (pg_phy_addr - kernel_pool.phy_addr_start) / PG_SIZE;
588     }
589     bitmap_set(&mem_pool->pool_bitmap, bit_idx, 0);
590 }
…略
```

函数 free_a_phy_page 接受 1 个参数，物理页框地址 pg_phy_addr，功能是根据物理页框地址 pg_phy_addr 在相应的内存池的位图清 0，此函数并不会改动页表。

函数的实现很容易，根据 pg_phy_addr 的值，判断它所属的物理内存池，算出与物理内存池的起始物理地址的差，使差再除以 PG_SIZE，所得的商便是在位图中的索引 bit_idx，最后调用 bitmap_set 在相应物理内存池中将 bit_idx 置为 0。

memory.c 就介绍完了，下面是重点，我们在 thread.c 中增加一些内容，见代码 15-32。

代码 15-32　（project/c15/i/thread/thread.c）

```
…略
 14 /* pid 的位图，最大支持 1024 个 pid */
 15 uint8_t pid_bitmap_bits[128] = {0};
 16
```

```
 17 /* pid 池 */
 18 struct pid_pool {
 19     struct bitmap pid_bitmap;  // pid 位图
 20     uint32_t pid_start;       // 起始 pid
 21     struct lock pid_lock;     // 分配 pid 锁
 22 }pid_pool;
…略
 56 /* 初始化 pid 池 */
 57 static void pid_pool_init(void) {
 58     pid_pool.pid_start = 1;
 59     pid_pool.pid_bitmap.bits = pid_bitmap_bits;
 60     pid_pool.pid_bitmap.btmp_bytes_len = 128;
 61     bitmap_init(&pid_pool.pid_bitmap);
 62     lock_init(&pid_pool.pid_lock);
 63 }
 64
 65 /* 分配 pid */
 66 static pid_t allocate_pid(void) {
 67     lock_acquire(&pid_pool.pid_lock);
 68     int32_t bit_idx = bitmap_scan(&pid_pool.pid_bitmap, 1);
 69     bitmap_set(&pid_pool.pid_bitmap, bit_idx, 1);
 70     lock_release(&pid_pool.pid_lock);
 71     return (bit_idx + pid_pool.pid_start);
 72 }
 73
 74 /* 释放 pid */
 75 void release_pid(pid_t pid) {
 76     lock_acquire(&pid_pool.pid_lock);
 77     int32_t bit_idx = pid - pid_pool.pid_start;
 78     bitmap_set(&pid_pool.pid_bitmap, bit_idx, 0);
 79     lock_release(&pid_pool.pid_lock);
 80 }
…略
309 /* 回收 thread_over 的 pcb 和页表，并将其从调度队列中去除 */
310 void thread_exit(struct task_struct* thread_over, bool need_schedule) {
311     /* 要保证 schedule 在关中断情况下调用 */
312     intr_disable();
313     thread_over->status = TASK_DIED;
314
315     /* 如果 thread_over 不是当前线程,
        就有可能还在就绪队列中，将其从中删除 */
316     if (elem_find(&thread_ready_list, &thread_over->general_tag)) {
317         list_remove(&thread_over->general_tag);
318     }
319     if (thread_over->pgdir) {     // 如是进程，回收进程的页表
320          mfree_page(PF_KERNEL, thread_over->pgdir, 1);
321     }
322
323     /* 从 all_thread_list 中去掉此任务 */
324     list_remove(&thread_over->all_list_tag);
325
326     /* 回收 pcb 所在的页，主线程的 pcb 不在堆中，跨过 */
327     if (thread_over != main_thread) {
328          mfree_page(PF_KERNEL, thread_over, 1);
329     }
330
331     /* 归还 pid */
332     release_pid(thread_over->pid);
333
334     /* 如果需要下一轮调度则主动调用 schedule */
335     if (need_schedule) {
336         schedule();
337         PANIC("thread_exit: should not be here\n");
338     }
339 }
340
341 /* 比对任务的 pid */
342 static bool pid_check(struct list_elem* pelem, int32_t pid) {
343     struct task_struct* pthread = elem2entry(struct task_struct, \
        all_list_tag, pelem);
344     if (pthread->pid == pid) {
345          return true;
```

```
346     }
347     return false;
348 }
349
350 /* 根据pid找pcb，若找到则返回该pcb，否则返回NULL */
351 struct task_struct* pid2thread(int32_t pid) {
352     struct list_elem* pelem = list_traversal(&thread_all_list, pid_check, pid);
353     if (pelem == NULL) {
354         return NULL;
355     }
356     struct task_struct* thread = elem2entry(struct task_struct,\
        all_list_tag, pelem);
357     return thread;
358 }
359
360 /* 初始化线程环境 */
361 void thread_init(void) {
362      put_str("thread_init start\n");
363
364     list_init(&thread_ready_list);
365     list_init(&thread_all_list);
366     pid_pool_init();
367
368  /* 先创建第一个用户进程:init */
369     process_execute(init, "init");
         // 放在第一个初始化，这是第一个进程,init进程的pid为1
370
371 /* 将当前main函数创建为线程 */
372     make_main_thread();
373
374     /* 创建idle线程 */
375     idle_thread = thread_start("idle", 10, idle, NULL);
376
377     put_str("thread_init done\n");
378 }
…略
```

这次我们改进了 pid 的分配，之前的 pid 只有分配，不能释放，前面已经介绍过僵尸进程的弊端了，也就是 pcb 和 pid 都不释放，咱们为了解决这个问题，把 pid 分配策略彻底改了。为了实现 pid 的管理，为它专门定义了 pid 池，池中包含了位图，由位图去管理 pid 的分配与释放。

第 15 行的 pid_bitmap_bits 是我们为 pid 池定义的位图，它为 128 字节，这说明咱们最大支持 1024 个 pid。

第 18～22 行定义的 struct pid_pool 是我们的 pid 池、并且生成了实例 pid_pool。

以下的 pid_pool_init 是初始化 pid_pool，对位图、起始 pid、锁进行了具体初始化工作，它在 therad_init 中被调用，也就是第 366 行，实现很简单，不用多说了。接下来的函数 allocate_pid 用于分配 pid，其内部操作已经是老生常谈了，函数 release_pid 用于释放 pid，函数实现也一样很老套了，不浪费大伙儿时间介绍了。

以上几个函数是为回收 pid 做出的改进，下面几个函数是为释放进程的 pcb 添加的新功能。

thread_exit 接受 2 个参数，待退出的任务 thread_over、是否要调度标记 need_schedule，功能是回收 thread_over 的 pcb 和页表，并将其从调度队列中去除。

函数实现中，先将 thread_over 的 status 置为 TASK_DIED，这表示该任务马上要结束生命周期。

当前线程肯定不在就绪队列当中，如果 thread_over 不是当前线程，就有可能在就绪队列中，因此在第 316 行，判断 thread_over 是否在就绪队列 thread_ready_list 中，如果是，就将其去掉。

第 319 行，如果 thread_over 是进程的话，就在下一行通过 mfree_page 回收其页目录表占用的 1 页框。

第 324 行，把 thread_over 从 all_thread_list 中去掉。

退出的线程有可能是主线程，而主线程的 pcb 并不是在堆中分配的，因此第 327 行对此特殊处理，对回收除主线程之外的任务的 pcb。

第 332 行通过 release_pid 释放了进程的 pid。

第 335 行通过 need_schedule 判断是否要调用 schedule 重新调度新进程。在后面您就会看到，我们在调用 thread_exit 时，有时候需要开始新调度，不用回到主调函数，有时候不需要新调度，调用 thread_exit

后还要回到主调函数中。

函数 pid_check 是函数 listr_traversal 的回调函数，它用于比对任务的 pid，找到特定 pid 的任务就返回。

函数 pid2thread 接受 1 个参数，任务的 pid，功能是根据 pid 找 pcb，若找到则返回该 pcb，否则返回 NULL，其原理是调用 list_traversal 遍历全部队列中的所有任务，通过回调函数 pid_check 过滤出特定 pid 的任务。

好啦，基础工作完成了，下节我们可以正式完成 wait 和 exit 了，下节见。

15.6.4 实现 wait 和 exit

前面为完成本节的内容做了很多铺垫，终于到了面对 wait 和 exit 的时刻了，有了它们，我们的父进程就能够获取子进程的返回值，并且，我们不用在程序中人为添加死循环 while(1) 来限制程序“边界”了。

Linux 中 exit 的系统调用是_exit，其原型是“void _exit(int status)”，其中 status 是返回的状态值，是子进程代入的参数。直接叫_exit 还真不习惯，为了具有亲和力，我们还是叫 exit 吧，接口的实现形式不变。

wait 的原型是“pid_t wait(int *status)”，其中 status 是父进程传入的地址，该地址空间用于接收子进程返回值，成功则返回子进程的 pid，失败则返回−1。

好，上代码，wait 和 exit 我们定义在 userprog/wait_exit.c 中，代码有点长，分为几部分，下面是第一部分，见代码 15-33-1。

代码 15-33-1 （project/c15/i/userprog/wait_exit.c）

```
…略
11 /* 释放用户进程资源:
12  * 1 页表中对应的物理页
13  * 2 虚拟内存池占物理页框
14  * 3 关闭打开的文件 */
15 static void release_prog_resource(struct task_struct* release_thread) {
16     uint32_t* pgdir_vaddr = release_thread->pgdir;
17     uint16_t user_pde_nr = 768, pde_idx = 0;
18     uint32_t pde = 0;
19     uint32_t* v_pde_ptr = NULL;      // v 表示 var，和函数 pde_ptr 区分
20
21     uint16_t user_pte_nr = 1024, pte_idx = 0;
22     uint32_t pte = 0;
23     uint32_t* v_pte_ptr = NULL;    // 加个 v 表示 var，和函数 pte_ptr 区分
24
25     uint32_t* first_pte_vaddr_in_pde = NULL;
       // 用来记录 pde 中第 1 个 pte 指向的物理页起始地址
26     uint32_t pg_phy_addr = 0;
27
28     /* 回收页表中用户空间的页框 */
29     while (pde_idx < user_pde_nr) {
30        v_pde_ptr = pgdir_vaddr + pde_idx;
31        pde = *v_pde_ptr;
32        if (pde & 0x00000001) {
          // 如果页目录项 p 位为 1，表示该页目录项下可能有页表项
33             first_pte_vaddr_in_pde = pte_ptr(pde_idx * 0x400000);
          // 一个页表表示的内存容量是 4MB，即 0x400000
34             pte_idx = 0;
35             while (pte_idx < user_pte_nr) {
36                v_pte_ptr = first_pte_vaddr_in_pde + pte_idx;
37                pte = *v_pte_ptr;
38                if (pte & 0x00000001) {
39        /* 将 pte 中记录的物理页框直接在相应内存池的位图中清 0 */
40                     pg_phy_addr = pte & 0xfffff000;
41                     free_a_phy_page(pg_phy_addr);
42                }
43                pte_idx++;
44             }
45           /* 将 pde 中记录的物理页框直接在相应内存池的位图中清 0 */
46           pg_phy_addr = pde & 0xfffff000;
47           free_a_phy_page(pg_phy_addr);
48        }
49        pde_idx++;
50     }
51
```

```
52      /* 回收用户虚拟地址池所占的物理内存*/
53      uint32_t bitmap_pg_cnt =\ (release_thread->userprog_vaddr.vaddr_bitmap.btmp_ bytes_len)
        / PG_SIZE;
54       uint8_t* user_vaddr_pool_bitmap = \
                  release_thread ->userprog_vaddr.vaddr_bitmap.bits;
55       mfree_page(PF_KERNEL, user_vaddr_pool_bitmap, bitmap_pg_cnt);
56
57       /* 关闭进程打开的文件 */
58       uint8_t fd_idx = 3;
59       while(fd_idx < MAX_FILES_OPEN_PER_PROC) {
60          if (release_thread->fd_table[fd_idx] != -1) {
61              sys_close(fd_idx);
62          }
63          fd_idx++;
64       }
65 }
```

函数 release_prog_resource 接受 1 个参数，待释放的任务 release_thread，功能是释放任务的资源，资源包括：页表中的物理页、虚拟内存池占物理页、关闭打开的文件。

函数中先完成的工作是回收页表中的物理页框，回收的方法有两种，一是按照虚拟内存池 pcb->userprog_vaddr 回收，检查位图中被置为 1 的 bit，计算出相应虚拟地址，逐位回收。另一种方法相对直接一点，直接遍历页表，如果页表的 p 位为 1，这说明已经分配了物理页框。第 1 种方法咱们已经在实现 fork 时为复制用户地址空间用过了，现在咱们尝试第二种方法。

第 17 行的变量 user_pde_nr 表示用户空间中 pde 的数量，其值为 768，pde_idx 表示 pde 的索引值，从 0 起。

第 21 行的变量 user_pte_nr 表示每个页表中 pte 的数据，其值为 1024，pte_idx 表示 pte 的索引值，从 0 起。

第 25 行的变量 first_pte_vaddr_in_pde 表示 pde 中第 0 个 pte 的地址，主要是用它来遍历页表中所有 pte。

第 29～50 行通过两层 while 循环回收页表中用户空间的页框，大伙儿能看到这，代码肯定是不需要细说了，大体上是在外层循环中判断页目录中的 pde，如果 pde 的 p 位为 1，表示该 pde 中“可能”会有页表，为什么说可能有呢？原因是回收内存空间时，页表中的 pte 很可能被回收干净了，但该页表所在的 pde 并不释放，也就是说 pde 中的页表地址还在，当初是为了减少页表的频繁变动而有意为之。一个页表能表示的内存范围是 1024*4KB=4MB，一个 pde 便表示一个页表，故我们可以根据当前是第几个 pde，即 pde_idx 的值，推算出虚拟地址范围，有了这个范围便足够了，无需精确到具体，只要知道虚拟地址对应于该物理页的起始地址便可。第 33 行的 first_pte_ vaddr_in_pde 通过 pte_ptr 函数获取第 pde_idx 个页表中第 0 个 pte 的虚拟地址。目的是通过虚拟地址遍历页表中的所有 pte。内层循环用来遍历每一个 pte，如果 pte 的 P 位为 1，表示已分配了物理页，将其通过 free_a_phy_page 回收。

接下来第 53～55 行是回收用户虚拟地址池所占的物理内存，最后第 58～64 行是关闭进程打开的文件，以上代码我想应该不用解释了。

下面是第二部分，见代码 15-33-2。

代码 15-33-2　（project/c15/i/userprog/wait_exit.c）

```
…略
67 /* list_traversal 的回调函数,
68    * 查找 pelem 的 parent_pid 是否是 ppid，成功返回 true，失败则返回 false */
69 static bool find_child(struct list_elem* pelem, int32_t ppid) {
70     /* elem2entry 中间的参数 all_list_tag 取决于 pelem 对应的变量名 */
71      struct task_struct* pthread = \
        elem2entry(struct task_struct, all_list_tag, pelem);
72      if (pthread->parent_pid == ppid) {
        // 若该任务的 parent_pid 为 ppid，返回
73           return true;   // list_traversal 只有在回调函数返回 true 时才会停止
                           // 继续遍历，所以在此返回 true
74      }
75      return false;     // 让 list_traversal 继续传递下一个元素
76 }
77
78      /* list_traversal 的回调函数,
79       * 查找状态为 TASK_HANGING 的任务 */
80       static bool find_hanging_child(struct list_elem* pelem, int32_t ppid) {
```

```
81     struct task_struct* pthread = elem2entry(struct task_struct,\
       all_list_tag, pelem);
82     if (pthread->parent_pid == ppid &&\
       pthread->status == TASK_HANGING) {
83         return true;
84     }
85     return false;
86 }
87
88 /* list_traversal 的回调函数,
89  * 将一个子进程过继给 init */
90   static bool init_adopt_a_child(struct list_elem* pelem, int32_t pid) {
91     struct task_struct* pthread = elem2entry(struct task_struct,\
     all_list_tag, pelem);
92     if (pthread->parent_pid == pid) { // 若该进程的 parent_pid 为 pid,返回
93         pthread->parent_pid = 1;
94     }
95     return false;      // 让 list_traversal 继续传递下一个元素
96 }
…略
```

函数 find_child 是 list_traversal 的回调函数，功能是查找 pelem 的 parent_pid 是否是 ppid，成功返回 true，失败则返回 false。函数实现挺直白的，就是找父进程 pid 为 ppid 的子进程，找到后返回 true，大伙儿自己看下吧。

函数 find_hanging_child 是专门找状态为 TASK_HANGING 的子进程，同上面的 find_child 类似，只是多了个状态判断。

函数 init_adopt_a_child 也是 list_traversal 的回调函数，功能是将 parent_pid 等于 pid 的进程过继给 init，使 init 作为该进程的父进程。实现也很简单，不说啦。下面看最后一部分，见代码 15-33-3。

代码 15-33-3 （project/c15/i/userprog/wait_exit.c）

```
…略
 98 /* 等待子进程调用 exit,将子进程的退出状态保存到 status 指向的变量.
 99  * 成功则返回子进程的 pid,失败则返回-1 */
100 pid_t sys_wait(int32_t* status) {
101     struct task_struct* parent_thread = running_thread();
102
103     while(1) {
104         /* 优先处理已经是挂起状态的任务 */
105         struct list_elem* child_elem = list_traversal(&thread_all_list,\
            find_hanging_child, parent_thread->pid);
106         /* 若有挂起的子进程 */
107         if (child_elem != NULL) {
108              struct task_struct* child_thread = \
                 elem2entry(struct task_struct, all_list_tag, child_elem);
109              *status = child_thread->exit_status;
110
111              /* thread_exit 之后,pcb 会被回收,因此提前获取 pid */
112              uint16_t child_pid = child_thread->pid;
113
114              /* 2 从就绪队列和全部队列中删除进程表项*/
115              thread_exit(child_thread, false);
             // 传入 false,使 thread_exit 调用后回到此处
116          /* 进程表项是进程或线程的最后保留的资源,至此该进程彻底消失了 */
117
118              return child_pid;
119         }
120
121         /* 判断是否有子进程 */
122         child_elem = list_traversal(&thread_all_list,\
            find_child, parent_thread->pid);
123         if (child_elem == NULL) {  // 若没有子进程,则出错返回
124              return -1;
125         } else {
126         /* 若子进程还未运行完成,即还未调用 exit,则将自己挂起,
            直到子进程在执行 exit 时将自己唤醒 */
127              thread_block(TASK_WAITING);
128         }
129     }
130 }
```

```
131
132  /* 子进程用来结束自己时调用 */
133  void sys_exit(int32_t status) {
134     struct task_struct* child_thread = running_thread();
135     child_thread->exit_status = status;
136     if (child_thread->parent_pid == -1) {
137         PANIC("sys_exit: child_thread->parent_pid is -1\n");
138     }
139
140     /* 将进程 child_thread 的所有子进程都过继给 init */
141     list_traversal(&thread_all_list, init_adopt_a_child, child_thread->pid);
142
143     /* 回收进程 child_thread 的资源 */
144     release_prog_resource(child_thread);
145
146     /* 如果父进程正在等待子进程退出，将父进程唤醒 */
147     struct task_struct* parent_thread = \
        pid2thread(child_thread->parent_pid);
148     if (parent_thread->status == TASK_WAITING) {
149         thread_unblock(parent_thread);
150     }
151
152     /* 将自己挂起，等待父进程获取其 status，并回收其 pcb */
153     thread_block(TASK_HANGING);
154 }
```

函数 sys_wait 接收一个参数，存储子进程返回值的地址 status，功能是等待子进程调用 exit，将子进程的退出状态保存到 status 指向的变量。

函数开头先调用 running_thread 获得当前任务，也就是父进程 parent_thread，接着是一个 while 循环，在第 103 行通过 list_traversal 在全部队列 thread_all_list 中遍历，通过回调函数 find_hanging_child 过滤出父进程 pid（parent_pid）为 parent_thread->pid，并且 status 为 TASK_HANGING 的子进程。此处是为了优先处理已经退出的进程。

如果有已退出的子进程，就在第 108～118 行开始做善后工作。先在第 109 行从子进程的 exit_status 中获取子进程的状态到 status 中，然后第 112 行获取子进程的 pid 到 child_pid 中，随后调用 thread_exit 把子进程从队列中删除，这里传给 thread_exit 的第二个参数是 false，即表示调用 thread_exit 后还要回来，并不是一去不回头，因为我们还要在第 118 行把子进程的 pid，即 child_pid 返回。

第 122 行，如果没有已退出的子进程，这时候再遍历一次查看是否有子进程，如果没有，就在第 124 行返回−1，如果有子进程，此时说明它的状态必然不是 TASK_HANGING，也就是说子进程尚未调用 exit，因此在第 127 行执行“thread_block(TASK_WAITING)”阻塞自己，直到子进程执行 exit 时把自己唤醒。下面看过函数 sys_exit 之后，大伙儿就知道子进程是怎样把它唤醒的了。

函数 sys_exit 接受一个参数，退出状态 status，此函数是子进程用来结束自己时调用。

函数开头先调用 running_thread 获得自己的 pcb，即 child_thread，随后将 status 存入自己 pcb 的 exit_status 中。

当前退出的进程有可能还有子进程，于是在第 141 行调用 list_traversal 遍历全部队列 thread_all_list，通过回调函数 init_adopt_a_child 将自己的子进程全部过继给 init。

第 144 行调用 release_prog_resource 释放自己除了 pcb 以外的资源，您懂的，pcb 中的 exit_status 父进程还没来收走呢，因此 pcb 得由父进程在调用 wait 获取其状态时再回收了。

第 147 行通过函数 pid2thread 获得自己的父进程 parent_thread，第 148 行判断父进程是否正在等待子进程退出，如果父进程正在等待自己，其状态 status 应该为 TASK_WAITING，于是在第 149 行通过“thread_unblock (parent_thread)”把父进程唤醒。

最后通过 thread_block 将自己挂起，并将自己的状态置为 TASK_HANGING，这样父进程便知道子进程已经退出了，可以获取退出状态值，并回收自己的 pcb。（说到这里一阵伤感凝重。）

好啦，代码部分就说完了，接下来就是添加系统调用 wait 和 exit，这个大伙儿不用说了吧，不单独贴代码了，下节咱们利用这些代码做点实事，本节到这就结束啦，大伙儿辛苦了。

15.6.5 实现 cat 命令

一直以来咱们缺少个查看文件的命令，在 Linux 中最著名的文件查看工具莫过于 cat 命令了，今天咱们实现一个简单的 cat，真的非常简单，不支持选项参数，只能查看普通文本文件。

目前我们系统中的系统调用如图 15-15 所示。

```
 7 enum SYSCALL_NR {
 8     SYS_GETPID,
 9     SYS_WRITE,
10     SYS_MALLOC,
11     SYS_FREE,
12     SYS_FORK,
13     SYS_READ,
14     SYS_PUTCHAR,
15     SYS_CLEAR,
16     SYS_GETCWD,
17     SYS_OPEN,
18     SYS_CLOSE,
19     SYS_LSEEK,
20     SYS_UNLINK,
21     SYS_MKDIR,
22     SYS_OPENDIR,
23     SYS_CLOSEDIR,
24     SYS_CHDIR,
25     SYS_RMDIR,
26     SYS_READDIR,
27     SYS_REWINDDIR,
28     SYS_STAT,
29     SYS_PS,
30     SYS_EXECV,
31     SYS_EXIT,
32     SYS_WAIT
33 };
```

▲图 15-15　系统调用

最后的 exit 和 wait 是我悄悄安装好的。由于我们已经实现 exit 和 wait，咱们现在不用在函数结束处用死循环“卡住”程序了。前面说过了，进程都会调用 wait，无论是进程自己调用 wait，还是由 C 运行库调用，现在咱们把 exit 加入到我们的简陋 C 运行库中，见代码 15-34。

代码 15-34　（project/c15/i/command/start.S）

```
 1 [bits 32]
 2 extern   main
 3 extern   exit
 4 section .text
 5 global _start
 6 _start:
 7     ;下面这两个要和 execv 中 load 之后指定的寄存器一致
 8     push  ebx      ;压入 argv
 9     push  ecx      ;压入 argc
10     call  main
11
12   ;将 main 的返回值通过栈传给 exit，gcc 用 eax 存储返回值，这是 ABI 规定的
13     push  eax
14     call exit
15     ;exit 不会返回
```

start.S 的主要变更是第 12 行，根据 ABI 规定，函数返回值是在 eax 寄存器中。在第 10 行调用 main 之后，第 13 行把 main 的返回值 eax 压栈，这是为第 14 行调用 exit 系统调用压入的参数，相当于 exit(eax)。到这用户进程差不多就结束了，后面是内核开始为其回收资源，父进程获取子进程的返回值，然后回收子进程 pcb。

下面是今天的主角，简易版 cat 的实现，它定义在 command/cat.c 中，见代码 15-35。

代码 15-35　（project/c15/i/command/cat.c）

```
1 #include "syscall.h"
2 #include "stdio.h"
3 #include "string.h"
4 int main(int argc, char** argv) {
5     if (argc > 2 || argc == 1) {
```

```
 6          printf("cat: only support 1 argument.\neg: cat filename\n");
 7          exit(-2);
 8     }
 9     int buf_size = 1024;
10     char abs_path[512] = {0};
11     void* buf = malloc(buf_size);
12     if (buf == NULL) {
13          printf("cat: malloc memory failed\n");
14          return -1;
15     }
16     if (argv[1][0] != '/') {
17          getcwd(abs_path, 512);
18          strcat(abs_path, "/");
19          strcat(abs_path, argv[1]);
20     } else {
21          strcpy(abs_path, argv[1]);
22     }
23     int fd = open(abs_path, O_RDONLY);
24     if (fd == -1) {
25          printf("cat: open: open %s failed\n", argv[1]);
26          return -1;
27     }
28     int read_bytes= 0;
29     while (1) {
30        read_bytes = read(fd, buf, buf_size);
31        if (read_bytes == -1) {
32             break;
33        }
34        write(1, buf, read_bytes);
35     }
36     free(buf);
37     close(fd);
38     return 66;
39 }
```

函数开头对参数个数判断，咱们的 cat 只支持 1 个参数，就是待查看的文件名。如果参数个数大于 2 或者没有参数，也就是 argc 为 1，那么输出报错后就调用“exit(−2)”退出，此处传入的状态值为−2。

接着通过 malloc 从堆中申请了 1024 字节的内存用作缓冲区 buf，512 字节的 abs_path 用于存储参数的绝对路径。

第 16～22 行处理参数文件的路径为绝对路径，之后存入到 abs_buf 中。

第 23 行通过 open 打开参数文件，第 29～35 行循环读取文件，然后通过 write 输出，直到 read 返回值为−1，也就是一直读到文件尾。

最后释放 buf 并关闭参数文件，把 66 作为返回值返回。

下面是编译脚本 compile.c，它实现的功能大伙儿都清楚了，具体见代码 15-36。

代码 15-36　（project/c15/i/command/compile.sh）

```
 1 ####  此脚本应该在 command 目录下执行
 2
 3 if [[ ! -d "../lib" || ! -d "../build" ]];then
 4       echo "dependent dir don\`t exist!"
 5       cwd=$(pwd)
 6       cwd=${cwd##*/}
 7       cwd=${cwd%/}
 8       if [[ $cwd != "command" ]];then
 9            echo -e "you\`d better in command dir\n"
10       fi
11       exit
12 fi
13
14 BIN="cat"
15 CFLAGS="-Wall -c -fno-builtin -W -Wstrict-prototypes \
16            -Wmissing-prototypes -Wsystem-headers"
17 LIBS="-I ../lib/ -I ../lib/kernel/ -I ../lib/user/ -I \
18            ../kernel/ -I ../device/ -I ../thread/ -I \
19            ../userprog/ -I ../fs/ -I ../shell/"
20 OBJS="../build/string.o ../build/syscall.o \
```

```
21             ../build/stdio.o ../build/assert.o start.o"
22 DD_IN=$BIN
23 DD_OUT="/home/work/my_workspace/bochs/hd60M.img"
24
25 nasm -f elf ./start.S -o ./start.o
26 ar rcs simple_crt.a $OBJS start.o
27 gcc $CFLAGS $LIBS -o $BIN".o" $BIN".c"
28 ld $BIN".o" simple_crt.a -o $BIN
29 SEC_CNT=$(ls -l $BIN|awk '{printf("%d", ($5+511)/512)}')
30
31 if [[ -f $BIN ]];then
32        dd if=./$DD_IN of=$DD_OUT bs=512 \
33        count=$SEC_CNT seek=300 conv=notrunc
34 fi
```

在 command 目录下执行 sh compile.sh 后，当前 command 目录下就生成了 cat 命令，其大小是 5476 字节，脚本中的 dd 命令会将其写入 hd60M.img 的第 300 个扇区以后。

有了 wait 之后，咱们要把相关的“while(1)”去掉，首先是 shell.c，见代码 15-37。

代码 15-37 （project/c15/i/shell/shell.c）

```
…略
155      } else {      // 如果是外部命令，需要从磁盘上加载
156          int32_t pid = fork();
157          if (pid) {        // 父进程
158               int32_t status;
159               int32_t child_pid = wait(&status);
         // 此时子进程若没有执行 exit,my_shell 会被阻塞，不再响应键入的命令
160               if (child_pid == -1) {
         // 按理说程序正确的话不会执行到这句，fork 出的进程便是 shell 子进程
161                    panic("my_shell: no child\n");
162               }
163               printf("child_pid %d, it's status: %d\n", child_pid, status);
164          } else {          // 子进程
165               make_clear_abs_path(argv[0], final_path);
166               argv[0] = final_path;
167               /* 先判断下文件是否存在 */
168               struct stat file_stat;
169               memset(&file_stat, 0, sizeof(struct stat));
170               if (stat(argv[0], &file_stat) == -1) {
171                    printf("my_shell: cannot access %s:
                    No such file or directory\n", argv[0]);
172                    exit(-1);
173                } else {
174                    execv(argv[0], argv);
175                }
176            }
177       }
…略
```

第 157～159 行，shell 派生出子进程后，父进程调用 wait 等待子进程的返回值。理论上外部命令就是 shell 的子进程，如果正常的话，不会执行到第 160～162 行的代码。第 163 行输出子进程 pid 及返回值。

在第 165～175 行的子进程中，如果外部命令的路径不存在，就在第 172 行调用“exit(−1)”返回，否则在第 174 行执行参数 argv[0]指向的进程。

好啦，下面我们把 cat 写入分区 sda 的根目录，见代码 15-38。

代码 15-38 （project/c15/i/kernel/main.c）

```
…略
21 int main(void) {
22     put_str("I am kernel\n");
23     init_all();
24
25 /*************    写入应用程序    *************/
26     uint32_t file_size = 5476;
27     uint32_t sec_cnt = DIV_ROUND_UP(file_size, 512);
```

```
28      struct disk* sda = &channels[0].devices[0];
29      void* prog_buf = sys_malloc(file_size);
30      ide_read(sda, 300, prog_buf, sec_cnt);
31      int32_t fd = sys_open("/cat", O_CREAT|O_RDWR);
32      if (fd != -1) {
33          if(sys_write(fd, prog_buf, file_size) == -1) {
34              printk("file write error!\n");
35              while(1);
36          }
37      }
38 /*************    写入应用程序结束   *************/
39      cls_screen();
40      console_put_str("[rabbit@localhost /]$ ");
41      thread_exit(running_thread(), true);
42      return 0;
43 }
44
45 /* init 进程 */
46 void init(void) {
47      uint32_t ret_pid = fork();
48      if(ret_pid) {  // 父进程
49          int status;
50          int child_pid;
51           /* init 在此处不停地回收僵尸进程 */
52           while(1) {
53              child_pid = wait(&status);
54              printf("I`m init, My pid is 1, I recieve a child,
                It`s pid is %d, status is %d\n", child_pid, status);
55           }
56      } else {       // 子进程
57          my_shell();
58      }
59      panic("init: should not be here");
60 }…略
```

除了在第 26～37 行写入 cat 命令外，还另外做了两件事，一件是主线程在第 41 行调用“thread_exit (running_thread(), true)”退出了，主线程使命结束后“请辞”了，让我们永远把它记在心里。另外就是在 init 进程中，第 52 行的 while 中，循环调用 wait，为过继给它的子进程做善后工作。

好啦，代码部分就这样了，根目录下已经有了文件 file1，下面可以测试 cat 命令。不过 file1 的内容就是两行“hello,world”，似乎未能尽兴，那我悄悄把 cat.c 写入到/dir1/下吧。

下面两张图是运行结果，请大伙儿过目。

图 15-16-1 中先执行了“ls –l”命令查看文件的写入结果，cat 命令已经在根目录下存在了。接着执行“cat file1”查看 file1 的内容，屏幕打印了两行“hello,world”，这是在很久很久以前咱们就写好的文件。shell 也输出了子进程的 pid 为 2，返回值为 66。下面调用 ps 命令显示任务列表，这时候熟悉的 main 线程已经没有了，由于之前各任务都在最后加入了死循环“while(1)”，所有任务都非常忙碌，idle 线程一直没机会运行，现在用 exit 替换了“while(1)”后，idle 线程都开始忙活了，这说明系统负载降下来了，大部分时间系统都处于挂起状态，只是中断信号让其“复活”反复运行。

接着用 ls 命令查看 dir1 目录，里面是我私下上传的 cat.c，然后为了查看错误情况下子进程的返回值，我用 cat 查看了一个不存在的路径“dir/cat.c”，该目录应该是“dir1/cat.c”，于是 shell 输出为子进程 pid 为 2，返回值为−1。下面继续看图 15-16-2。

图 15-16-2 所示是已经执行“cat dir1/cat.c”的结果，由于屏幕比较小，所以输入的命令被 cat.c 的内容覆盖了。父进程 shell 输出子进程 pid 是 2，返回值为 66。最后执行了无参数的 cat 命令，父进程 shell 输出子进程 pid 是 2，返回值为−2。

另外，大伙儿应该发现了，咱们任务的 pid 始终是 2，这说明已逝任务的 pid 正常释放了，之前 main 线程的 pid 为 2，它退出后，pid 就空出来了，不断分配给新的子进程，子进程又不断释放该 pid，所以该 pid 始终可用。

好啦，本节到这就结束了，感谢大伙儿收看。

```
Bochs x86 emulator, http://bochs.sourceforge.net/
[rabbit@localhost /]$ ls -l
total: 168
d  0  168  .
d  0  168  ..
-  1  24  file1
d  2  96  dir1
-  5  4777  prog_no_arg
-  6  5307  prog_arg
-  7  5476  cat
[rabbit@localhost /]$ ./cat file1
hello,world
hello,world
child_pid 2, it's status: 66
[rabbit@localhost /]$ ps
PID             PPID            STAT            TICKS           COMMAND
1               NULL            WAITING         84              init
3               NULL            BLOCKED         1157166         idle
4               1               RUNNING         167             init
[rabbit@localhost /]$ ls dir1
. .. dir11 cat.c
[rabbit@localhost /]$ cat dir/cat.c
cannot access //dir/cat.c: Not a directory, subpath /dir is`t exist
cat: open: open dir/cat.c failed
child_pid 2, it's status: -1
[rabbit@localhost /]$
IPS: 23.550M
```

▲图 15-16-1　cat 命令运行结果 1

```
Bochs x86 emulator, http://bochs.sourceforge.net/
      strcpy(abs_path, argv[1]);
   }
   int fd = open(abs_path, O_RDONLY);
   if (fd == -1) {
      printf("cat: open: open %s failed\n", argv[1]);
      return -1;
   }
   int read_bytes= 0;
   while (1) {
      read_bytes = read(fd, buf, buf_size);
      if (read_bytes == -1) {
         break;
      }
      write(1, buf, read_bytes);
   }
   free(buf);
   close(fd);
   return 66;
}
child_pid 2, it's status: 66
[rabbit@localhost /]$ cat
cat: only support 1 argument.
eg: cat filename
child_pid 2, it's status: -2
[rabbit@localhost /]$ c
IPS: 24.379M
```

▲图 15-16-2　cat 命令运行结果 2

15.7 管道

本节我们将实现管道系统调用，有了该功能后，我们可以支持父子进程通信，并且在 shell 中支持管道操作。

15.7.1 管道的原理

进程虽然是独立运行的个体，但它们之间有时候需要协作才能完成一项工作，比如有两个进程需要同步数据，进程 A 把数据准备好后，想把数据发往进程 B，进程 B 必须被提前通知有数据即将到来，或者进程 A 想发送信号给进程 B，以控制进程 B 的运行模式，再或者数据被多个进程共享时，数据变更后应该被所有进程看到，总之诸如此类的需求很多，操作系统必须要实现进程间的相互通信。

进程间通信方式有很多种，有消息队列、共享内存、socket 网络通信等，还有一种就是管道。鉴于能力有限，这里只打算实现管道，下面咱们一块儿讨论下管道的原理。

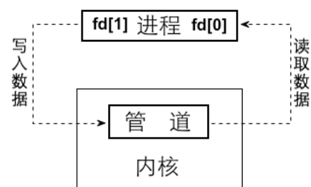
管道是进程间通信的方式之一，在 Linux 中一切皆文件，因此管道也被视为文件，只是该文件并不存

在于文件系统上，而是只存在于内存中。既然是文件，管道就要按照文件操作的函数来使用，因此也要使用 open、close、read、write 等方法来操作管道。管道通常被多个进程共享，而且存在于内存之中，因此共享的原理是所有进程在地址空间中都可以访问到它，所以您肯定猜到了，管道其实就是内核空间中的内存缓冲区。当然，进程间通信也可以通过文件系统，也就是说多个进程可以共同读写磁盘上的同一个文件来实现数据共享，但毕竟比较慢。

管道是用于存储数据的中转站，当某个进程往管道中写入数据后，该数据很快就会被另一个进程读取，之后可以用新的数据覆盖老数据，继续被别的进程读取，因此管道属于临时存储区，其中的数据在读取后可被清除。按理说，是存储区就应该有个空间大小，它的空间该多大才合适呢？似乎这取决于所写入的数据量，但数据量可大可小，没有上限，可以大到无穷，也可以小到 1 字节。可是，如果缓冲区小了就会丢数据，大了又无止境，多大的物理内存都不够，很难给个具体的大小。生活中给别人买衣服是最头疼的事了，因为就怕买大了或买小了，即使是对方穿着不合身，人家也不好意思让你给退了，好纠结……因此最好给他们买有弹力的衣服。管道也需要这种"弹力"的缓冲区，但这种"弹力"并不是指缓冲区可大可小，而是指一种可以写入无穷无尽的数据而不会有数据丢失的策略。卖了这么大的关子，其实这个弹力就是指"环形缓冲区"。

管道是个环形缓冲区，我们在之前介绍生产者消费者问题时已经使用过环形缓冲区了，就是咱们的 ioqueue，键盘输入缓冲区就是用它来实现的，想到这似乎觉得很欣慰，毕竟学习成本少了一些。回顾一下，对环形缓冲区的维护，主要是协调好数据读写的两个指针，以及生产者、消费者的休眠时机。环形缓冲区中一个指针用于读数据，另一个用于写数据。当缓冲区已满时，生产者要睡眠，并在睡眠前唤醒消费者，当缓冲区为空时，消费者要睡眠，并在睡眠前唤醒生产者。当缓冲区满或空时，使一方休眠，这是保证数据不丢失的方法。管道其实就是典型的生产者和消费者问题，有关这方面的介绍请参阅前面章节的相关内容。

管道有两端，一端用于从管道中读入数据，另一端用于往管道中写入数据。这两端使用文件描述符的方式来读取，故进程创建管道实际上是内核为其返回了用于读取管道缓冲区的文件描述符，一个描述符用于读，另一个描述符用于写。通常情况下是用户进程为内核提供一个长度为 2 的文件描述符数组，内核会在该数组中写入管道操作的两个描述符，假设数组名为 fd，那么 fd[0] 用于读取管道，fd[1] 用于写入管道，进程与管道的读写关系如图 15-17 所示。



▲图 15-17　管道

您看到了，进程创建了管道之后，自己往管道中写数据，然后自己再把数据从管道中读出来，这没什么实际意义，而且管道还白白占用了内核的空间，违背了"进程间"通信的初心。因此通常的用法是进程在创建管道之后，马上调用 fork，克隆出一个子进程，子进程完全继承了父进程的一切，也就是说和父进程一模一样，因此也继承了管道的描述符，这为父子进程通信提供了保证。父进程 fork 出子进程后，文件描述符的关系如图 15-18 所示。

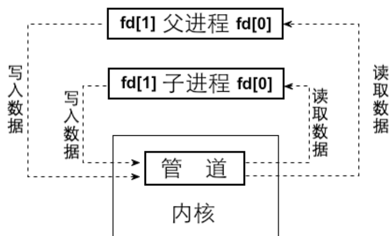
您看，父子进程完全一样，因此父子进程都可以通过文件描述符 fd[1] 向管道中写数据，通过文件描述符 fd[0] 从管道中读取数据。对管道的操作同对普通文件是一样的，比如父进程往管道中读写数据后，文件要更新读写的位置指针，父子进程的描述符指向的是相同的文件，管道也被视为文件，因此子进程再操作管道时，是从父进程管道操作之后的新位置处开始读写的，子进程操作管道之后文件指针也会更新，父进程也会在新位置处继续读写管道，总之父子进程指向同一个管道，实现了父子进程间的通信。

一般情况下，父子进程中都是一个读数据，一个写数据，并不会存在一方又读又写的情况，因此在父子进程中会分别关掉不使用的管道描述符。比如父进程负责往管道中写数据，它只需要 fd[1] 描述符，因此只可以通过 close 系统调用关闭 fd[0]。子进程负责从管道中读数据，它只需要 fd[0] 描述符，因此只可以通过 close 系统调用关闭 fd[1]。这时它们与管道的关系如图 15-19 所示，这也是管道操作中较常用的做法。

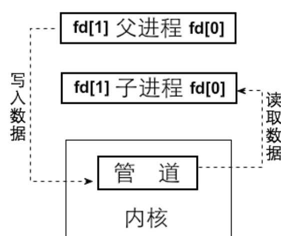
管道分为两种：匿名管道和命名管道，从概念上就可以知道，这是按照管道是否有名称来划分的。以上说的管道便是匿名管道，它没有名字。由于没有名字，匿名管道在创建之后只能通过内核为其返回的文件描述符来访问，此管道只对创建它的进程及其子进程可见，对其他进程不可见，因此除父子进程之外的其他进程便不知道此管道的存在，故匿名管道只能局限用于父子进程间的通信。

有名管道是专门为解决匿名管道的局限性而生的，在 Linux 中可以通过命令 mkfifo 来创建命名管道，

成功创建之后便会在文件系统上存在个管道文件，这使得该管道对任何进程都“可见”，因此多个进程即使没有父子关系也都通过访问该管道文件进行通信。



▲图 15-18 父子进程与管道



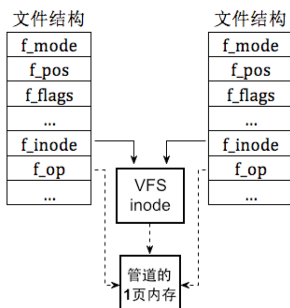
▲图 15-19 父子进程间通常的管道操作

有关管道的内容咱们就介绍到这里，目前只打算实现匿名管道，下一节咱们设计它的实现方式。

15.7.2 管道的设计

Linux 除了支持标准的文件系统 ext2、ext3、ext4 外，还支持其他文件系统，如 reiserfs、nfs 和 Windows 的 ntfs 等。为了向上提供统一的接口，Linux 加了一层中间层——VFS，即 Virtual File System，虚拟文件系统，向用户屏蔽了各种实现的细节，用户只和 VFS 打交道。

看看 Linux 是怎样处理管道的。管道对于 Linux 来说也是文件，因此它也需要用文件相关的数据结构来处理管道，Linux 是利用现有的文件结构和 VFS 索引结点的 inode 共同完成管道的，并没有单独为管道创建新的数据结构，结构示意如图 15-20 所示。



▲图 15-20 Linux 文件结构与 vfs 实现的管道

文件结构中的 f_inode 指向 VFS 的 inode，该 inode 指向 1 个页框大小的内存区域，该区域便是管道用于存储数据的内存空间。也就是说，Linux 的管道大小是 4096 字节。

f_op 用于指向操作（OPeration）方法，也就是说，不同的操作对象有不同的操作方法，针对不同的操作对象，Linux 会把 f_op 指向不同的操作例程。对于管道来说，f_op 会指向 pipe_read 和 pipe_write，pipe_read 会从管道的 1 页内存中读取数据，pipe_write 会往管道的 1 页内存中写入数据。

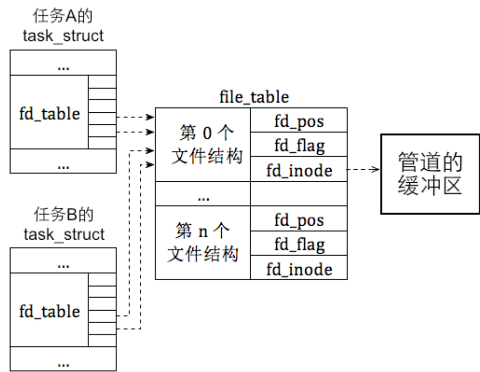
了解了 Linux 对管道的处理后，我们决定“部分地效仿”这种做法，注意这里强调的是“部分地效仿”，因为我们根本没法和 Linux 比，理由是我们只支持自己的文件系统，完全不需要用 VFS 这个中间层，并且只支持硬盘操作，完全不需要 f_op 来指定不同的操作方法。总之 Linux 太强大了，它甩我们几十条街不止，根本就没法和它比，完全不在同一个次元上，很多管理结构我们都不存在，因此只要实现其思路就好了。下面看看如何在咱们的系统中实现管道。

咱们的文件结构不像 Linux 文件结构那么丰满，咱们仅包括三个成员，fd_pos 用于表示文件读写位置，fd_flags 用于表示文件操作方式，fd_inode 用于表示文件的 inode 指针。按理说这三个成员的作用已经固定了，但单独为实现管道再添加个额外的数据结构就有些浪费了，再者 Linux 也是整合了现有资源实现的管道，咱们也可以复用现有的文件结构。

文件结构的成员名称已经是固定的了，再改变的话成本太高，现在也不想增加额外的成员，因此只能把成员的作用改变。在这之前，文件结构对应一个普通文件或目录的 inode，现在多了管道这种新的文件类型，而管道不需要 inode，那如何在文件结构中识别管道，而不是误把它当成一般的 inode 来处理呢？看来咱们得想办法在文件结构中为管道加个标志。这里的方法是把 fd_flags 成员动动手脚，如果此文件结构对应的是管道，那么 fd_flags 的值将是 0xFFFF，不再是 O_RDONLY、O_WRONLY 等值。另外，管道得有个存储数据的内存缓冲区，因此咱们把文件结构中的 fd_inode 指向管道的内存缓冲区，至于 fd_pos 嘛，咱们就把它用于此管道的打开数。经过复用以上三个成员，咱们的文件结构依然能够满足管道的需求。

文件是通过文件描述符访问的，此文件描述符是指向 pcb 中 fd_table 数组的下标，数组元素的值是我

们全局的文件表 file_table 的下标，该下标对应的文件结构 struct file 才是文件的描述信息。要想实现任意进程间通信，必须使它们访问公共的内存区域才行，具体到文件系统来说，无论进程的文件描述符是多少，只要使任意进程的文件描述符所指向的、位于 file_table 中的文件结构是同一个就行了，无论该文件结构中的 fd_inode 指向哪里，它肯定能被共享，毕竟咱们的功能比较简单，没必要像 Linux 那样使多个文件结构的 fd_inode 指向同一个缓冲区。综上所述，咱们实现管道的思路如图 15-21 所示。



▲图 15-21　管道的实现

图 15-21 中任务 A 和任务 B 中各画了两个箭头指向文件结构，实际上用于操作管道的文件描述符有两个，一个用于从管道中读取数据，另一个用于往管道中写入数据。

理论上，由于有了环形缓冲区，并且父子进程是并行运行，父子进程可以无限量地传递数据，因为无论生产者是父进程，还是子进程，当管道的环形缓冲区满了，生产者进程会被阻塞并唤醒消费者，作为另一方的消费者进程会从管道中继续获取数据，然后唤醒生产者，当管道为空时，消费者进程又会阻塞并唤醒生产者，总之无论管道的环形缓冲区多大，都能保证无损传输无限大的数据。但是，将来咱们会在命令行中支持管道操作符“|”，比如允许在 shell 中键入命令“a|b|c|d”，为了实现简单，目前管道的实现方式是按照从左到右逐个执行的，即管道中的命令并未实现并行，因此虽然用到了环形缓冲区，但实际上它并未真正发挥出其优势，也就是说，当管道的环形缓冲区满了后，生产者进程休眠，只要生产者未退出，后续的进程就没机会执行，因此无法作为管道的消费者，从而生产者永远休眠没机会被唤醒。对于消费者进程也是，如果管道空了，消费者也会永远阻塞，因为生产者已经退出了，消费者进程没有机会被唤醒。而且在 shell 中输入的管道命令理论上是无限多的，像“a|b|c|d”这是 4 级命令，“a|b|c|d|e|f”就是 6 级命令，上级命令的输出作为下级命令的输入，为了保证数据完整性，管道命令中两两相邻的命令要共用一个管道，如 a 和 b 共用一个管道，b 和 c 共用另一个管道，否则全部命令只共用一个管道的话，一定会破坏管道环形缓冲区中的数据。怎么办呢？一是本着说清楚管道的原理，二是减少实现的难度，考虑再三，这里采用一种折中的方法。为避免进程无限休眠的情况，我们让生产者和消费者每次只读写“适量”的数据，避免环形缓冲区满或空的情况，这样生产者或消费者进程就不会阻塞了。估计您猜到了，这个“适量”对于生产者来说是指环形缓冲区中可用的剩余空间大小，对于消费者来说是指环形缓冲区中的数据量。所以，如果在命令行中支持管道操作符，咱们的管道是有缺陷的，它所能传递的最大数据量是环形缓冲区的大小减一。

如果您能接受这种缺陷的话，咱们就带着这种遗憾继续下一节，感谢您的宽容。

15.7.3　管道的实现

在 Linux 中创建管道的方法是系统调用 pipe，其原型是“int pipe(int pipefd[2])”，成功返回 0，失败返回−1，其中 pipefd[2]是长度为 2 的整型数组，用来存储系统返回的文件描述符，文件描述符 fd[0]用于读取管道，fd[1]用于写入管道。下面咱们就按照此接口来实现。

本次在 ioqueue.c 中增加了函数 ioq_length，见代码 15-39。

代码 15-39　（project/c15/j/device/ioqueue.c）

```
…略
87 /* 返回环形缓冲区中的数据长度 */
88 uint32_t ioq_length(struct ioqueue* ioq) {
89     uint32_t len = 0;
90     if (ioq->head >= ioq->tail) {
91         len = ioq->head - ioq->tail;
92     } else {
93         len = bufsize - (ioq->tail - ioq->head);
94     }
95     return len;
96 }
…略
```

函数 ioq_length 接受 1 个参数，环形缓冲区 ioq，功能是返回环形缓冲区中的数据长度。原理还是很简单的，就不多说了。

有关管道的函数定义在 shell/pipe.c 中，这是本节新增的文件，下面是详细的代码，见代码 15-40。

代码 15-40 （project/c15/j/shell/pipe.c）

```
…略
 8 /* 判断文件描述符 local_fd 是否是管道 */
 9 bool is_pipe(uint32_t local_fd) {
10     uint32_t global_fd = fd_local2global(local_fd);
11     return file_table[global_fd].fd_flag == PIPE_FLAG;
12 }
13
14 /* 创建管道，成功返回 0，失败返回-1 */
15 int32_t sys_pipe(int32_t pipefd[2]) {
16     int32_t global_fd = get_free_slot_in_global();
17
18     /* 申请一页内核内存做环形缓冲区 */
19     file_table[global_fd].fd_inode = get_kernel_pages(1);
20
21     /* 初始化环形缓冲区 */
22     ioqueue_init((struct ioqueue*)file_table[global_fd].fd_inode);
23     if (file_table[global_fd].fd_inode == NULL) {
24          return -1;
25     }
26
27     /* 将 fd_flag 复用为管道标志 */
28     file_table[global_fd].fd_flag = PIPE_FLAG;
29
30     /* 将 fd_pos 复用为管道打开数 */
31     file_table[global_fd].fd_pos = 2;
32     pipefd[0] = pcb_fd_install(global_fd);
33     pipefd[1] = pcb_fd_install(global_fd);
34     return 0;
35 }
36
37 /* 从管道中读数据 */
38 uint32_t pipe_read(int32_t fd, void* buf, uint32_t count) {
39     char* buffer = buf;
40     uint32_t bytes_read = 0;
41     uint32_t global_fd = fd_local2global(fd);
42
43     /* 获取管道的环形缓冲区 */
44     struct ioqueue* ioq = (struct ioqueue*)file_table[global_fd].fd_inode;
45
46     /* 选择较小的数据读取量，避免阻塞 */
47     uint32_t ioq_len = ioq_length(ioq);
48     uint32_t size = ioq_len > count ? count : ioq_len;
49     while (bytes_read < size) {
50        *buffer = ioq_getchar(ioq);
51        bytes_read++;
52        buffer++;
53     }
54     return bytes_read;
55 }
56
57 /* 往管道中写数据 */
58 uint32_t pipe_write(int32_t fd, const void* buf, uint32_t count) {
59     uint32_t bytes_write = 0;
60     uint32_t global_fd = fd_local2global(fd);
61     struct ioqueue* ioq = (struct ioqueue*)file_table[global_fd].fd_inode;
62
63     /* 选择较小的数据写入量，避免阻塞 */
64     uint32_t ioq_left = bufsize - ioq_length(ioq);
65     uint32_t size = ioq_left > count ? count : ioq_left;
66
67     const char* buffer = buf;
68     while (bytes_write < size) {
69       ioq_putchar(ioq, *buffer);
70       bytes_write++;
```

```
71         buffer++;
72      }
73      return bytes_write;
74 }
```

先看下代码第二个函数 sys_pipe，它接受 1 个参数，存储管道文件描述符的数组 pipefd，功能是创建管道，成功后描述符 pipefd[0] 可用于读取管道，pipefd[1] 可用于写入管道，然后返回值为 0，否则返回−1。

函数先调用 get_free_slot_in_global 从 file_table 中获得可用的文件结构空位下标，记为 global_fd，然后第 19 行为该文件结构中的 fd_inode 分配一页内核内存做管道的环形缓冲区。接着第 22 行调用 ioqueue_init 初始化环形缓冲区。

第 28 行将该文件结构的 fd_flag 置为宏 PIPE_FLAG，宏 PIPE_FLAG 定义在 pipe.h 中，代码是“#define PIPE_FLAG 0xFFFF”，正如我们在设计阶段所说的，复用了文件结构中的 fd_flag 成员，把该值置为 0xFFFF 来表示此文件结构对应的是管道。

接着第 31 行把 fd_pos 置为 2，表示有两个文件描述符对应这个管道，这两个文件描述符是第 32～33 行通过 pcb_fd_install 来安装的，返回的描述符分别存储到 pipefd[0]和 pipefd[1]中，我们分别用它们来读取和写入管道。最后通过 return 返回 0，管道创建成功。

现在返回去说第一个函数 is_pipe，它接受 1 个参数，文件描述符 local_fd，也就是 pcb 中数组 fd_table 的下标，功能是判断文件描述符 local_fd 是否是管道。判断的原理是先找出 local_fd 对应的 file_table 中的下标 global_fd，然后判断文件表 file_talbe[global_fd]的 fd_flag 的值是否为 PIPE_FLAG。

函数 pipe_read 接受 3 个参数，文件描述符 fd、存储数据的缓冲区 buf、读取数据的数量 count，功能是从文件描述符 fd 中读取 count 字节到 buf。

第 41 行获得了 fd 对应的 file_table 中的下标 global_fd，第 44 行获得了相应文件结构中的环形缓冲区。

第 47～48 行根据缓冲区中数据量 ioq_len 和待读取的数据量 count 的大小，选择两者中较小的值作为读取的实际数据量 size，size 就是咱们在上节中所说的“适量”。第 49～53 行通过 while 循环调用 ioq_getchar 逐字节完成读取。

函数 pipe_write 功能是把缓冲区 buf 中的 count 个字节写入管道对应的文件描述符 fd。其实现同 pipe_read 雷同，不说了。

管道的操作也是通过文件系统，因此要修改相关文件系统的代码，如代码 15-41 所示。

代码 15-41　（project/c15/j/fs/fs.c）

```
…略
375 /* 关闭文件描述符 fd 指向的文件，成功返回 0，否则返回-1 */
376 int32_t sys_close(int32_t fd) {
377     int32_t ret = -1;   // 返回值默认为-1，即失败
378     if (fd > 2) {
379          uint32_t global_fd = fd_local2global(fd);
380          if (is_pipe(fd)) {
381               /* 如果此管道上的描述符都被关闭，释放管道的环形缓冲区 */
382               if (--file_table[global_fd].fd_pos == 0) {
383                    mfree_page(PF_KERNEL, file_table[global_fd].fd_inode, 1);
384                    file_table[global_fd].fd_inode = NULL;
385               }
386               ret = 0;
387           } else {
388               ret = file_close(&file_table[global_fd]);
389           }
390           running_thread()->fd_table[fd] = -1; // 使该文件描述符位可用
391     }
392     return ret;
393 }
394
395 /* 将 buf 中连续 count 个字节写入文件描述符 fd，
    成功则返回写入的字节数，失败返回-1 */
396 int32_t sys_write(int32_t fd, const void* buf, uint32_t count) {
397     if (fd < 0) {
398          printk("sys_write: fd error\n");
399          return -1;
```

```
400     }
401     if (fd == stdout_no) {
402          /* 标准输出有可能被重定向为管道缓冲区，因此要判断 */
403          if (is_pipe(fd)) {
404               return pipe_write(fd, buf, count);
405          } else {
406              char tmp_buf[1024] = {0};
407              memcpy(tmp_buf, buf, count);
408              console_put_str(tmp_buf);
409              return count;
410          }
411      } else if (is_pipe(fd)){        /* 若是管道就调用管道的方法 */
412          return pipe_write(fd, buf, count);
413      } else {
414          uint32_t _fd = fd_local2global(fd);
415          struct file* wr_file = &file_table[_fd];
416          if (wr_file->fd_flag & O_WRONLY || wr_file->fd_flag & O_RDWR) {
417              uint32_t bytes_written  = file_write(wr_file, buf, count);
418              return bytes_written;
419          } else {
420              console_put_str("sys_write: not allowed to write file
                 without flag O_RDWR or O_WRONLY\n");
421              return -1;
422          }
423     }
424 }
425
426    /* 从文件描述符 fd 指向的文件中读取 count 个字节到 buf，
      若成功则返回读出的字节数，到文件尾则返回-1 */
427    int32_t sys_read(int32_t fd, void* buf, uint32_t count) {
428     ASSERT(buf != NULL);
429     int32_t ret = -1;
430     uint32_t global_fd = 0;
431     if (fd < 0 || fd == stdout_no || fd == stderr_no) {
432          printk("sys_read: fd error\n");
433     } else if (fd == stdin_no) {
434         /* 标准输入有可能被重定向为管道缓冲区，因此要判断 */
435         if (is_pipe(fd)) {
436              ret = pipe_read(fd, buf, count);
437         } else {
438             char* buffer = buf;
439             uint32_t bytes_read = 0;
440             while (bytes_read < count) {
441                *buffer = ioq_getchar(&kbd_buf);
442                bytes_read++;
443                buffer++;
444             }
445             ret = (bytes_read == 0 ? -1 : (int32_t)bytes_read);
446         }
447     } else if (is_pipe(fd)) {     /* 若是管道就调用管道的方法 */
448         ret = pipe_read(fd, buf, count);
449     } else {
450         global_fd = fd_local2global(fd);
451         ret = file_read(&file_table[global_fd], buf, count);
452     }
453     return ret;
454 }
…略
```

关闭文件时，描述符 fd 对应的可能是管道，因此在函数 sys_close 中，我们在第 380～386 行加入了管道的处理，第 380 行通过函数 is_pipe(fd)判断关闭的文件描述符是否是管道，如果是就在第 382 行将相应文件结构的 fd_pos 减 1，如果减 1 后的值为 0，这说明没有文件描述符打开它了，所以在第 383 行调用 mfree_page 将管道环形缓冲区占用的 1 页内核页框释放。随后在第 384 行将相应文件结构中的 fd_inode 置为 NULL。

写入文件时，有可能写入的是管道，因此函数 sys_write 也做出了改动，在第 401 行处理标准输出的代码块中，第 403 行判断，如果标准输出是管道，这说明标准输出被重定向了（以后我们实现 shell 中管道操作就会涉及到重定向），就调用 pipe_write 方法写管道。第 411 行，如果 fd 不是标准描述符（标准输入、标准输出等），依然要通过 is_pipe 判断其是否是管道，如果是，就调用 pipe_write 方法写管道。

读入文件时，有可能读入的是管道，因此函数 sys_read 加入了对管道的处理。标准输入有可能被重定向，因此第 435 行调用 is_pipe 对此情况判断，如果确实是重定向了，就调用 pipe_read 读取管道。第 447～452 行是处理非标准描述符的代码，第 447 行判断如果是管道，就在第 448 行调用 pipe_read 完成。

管道是由父子进程共享的，因此在 fork 时也要增加管道的打开数，见代码 15-42。

代码 15-42　（project/c15/j/ userprog/fork.c）

```
…略
114 /* 更新 inode 打开数 */
115 static void update_inode_open_cnts(struct task_struct* thread) {
116     int32_t local_fd = 3, global_fd = 0;
117     while (local_fd < MAX_FILES_OPEN_PER_PROC) {
118        global_fd = thread->fd_table[local_fd];
119        ASSERT(global_fd < MAX_FILE_OPEN);
120        if (global_fd != -1) {
121           if (is_pipe(local_fd)) {
122              file_table[global_fd].fd_pos++;
123           } else {
124              file_table[global_fd].fd_inode->i_open_cnts++;
125           }
126        }
127        local_fd++;
128     }
129 }
…略
```

在函数 update_inode_open_cnts 的第 121 行，调用 is_pipe 判断是否为管道，如果是，就在第 122 行将对应文件结构的 fd_pos 加 1。

由于有了管道，程序退出时也要考虑相应的处理，见代码 15-43。

代码 15-43　（project/c15/j/ userprog/wait_exit.c）

```
…略
17 static void release_prog_resource(struct task_struct* release_thread) {
…略
59      /* 关闭进程打开的文件 */
60      uint8_t local_fd = 3;
61      while(local_fd < MAX_FILES_OPEN_PER_PROC) {
62         if (release_thread->fd_table[local_fd] != -1) {
63            if (is_pipe(local_fd)) {
64               uint32_t global_fd = fd_local2global(local_fd);
65               if (--file_table[global_fd].fd_pos == 0) {
66                  mfree_page(PF_KERNEL, file_table[global_fd].fd_inode, 1);
67                  file_table[global_fd].fd_inode = NULL;
68               }
69            } else {
70               sys_close(local_fd);
71            }
72         }
73         local_fd++;
74     }
75 }
```

如果程序退出时忘记关闭打开的文件或管道，在函数 release_prog_resource 中要关闭它们。第 63 行判断关闭的若是管道，就在第 65 行将对应文件结构的 fd_pod 减 1，如果减 1 后的值为 0，这说明没有进程再打开此管道了，此管道没用了，在第 66 行调用 mfree_page 回收管道环形缓冲区占用的一页内核页框。

另外，pipe 的系统调用我就悄悄添加了。

好啦，涉及的相关代码就改完了，本节到这结束。

15.7.4　利用管道实现进程间通信

本节咱们编写用户进程，在用户程序中创建管道来验证父子进程间的通信功能，见代码 15-44。

代码 15-44　（project/c15/j/command/prog_pipe.c）

```
1 #include "stdio.h"
2 #include "syscall.h"
3 #include "string.h"
4 int main(int argc, char** argv) {
```

```
 5      int32_t fd[2] = {-1};
 6      pipe(fd);
 7      int32_t pid = fork();
 8      if(pid) {      // 父进程
 9          close(fd[0]);  // 关闭输入
10          write(fd[1], "Hi, my son, I love you!", 24);
11          printf("\nI`m father, my pid is %d\n", getpid());
12          return 8;
13      } else {
14          close(fd[1]);  // 关闭输出
15          char buf[32] = {0};
16          read(fd[0], buf, 24);
17          printf("\nI`m child, my pid is %d\n", getpid());
18          printf("I`m child, my father said to me: \"%s\"\n", buf);
19          return 9;
20      }
21 }
```

prog_pipe.c 是咱们的测试用例，主要用来演示管道的功能，另外说明一下，主函数中的参数 argc 和 argv 并未用上。

函数开头先定义了数组 fd[2]，它用来存储管道返回的两个文件描述符。接着调用“pipe(fd)”创建管道，此时数组 fd 中已经是管道的两个描述符，我们用 fd[0]读管道，fd[1]写管道。接着调用 fork 派生子进程。父进程负责写管道，子进程读管道，因此在父进程代码中，第 9 行通过 close 关闭 fd[0]，然后调用 write 系统调用写入字符串“Hi, my son, I love you!”，父爱如山，满满正能量。然后调用 printf 输出“\nI`m father, my pid is…”，最后返回 8，父进程结束。

子进程通过 close 关闭 fd[1]，接着定义 32 字节的缓冲区 buf，然后调用 read 从 fd[0]中读取管道数据。然后第 17 行输出“\nI`m child, my pid is…”，接着第 18 行输出父进程对自己说的话，最后返回 9 子进程结束。

用户进程很简单，介绍完了，编译脚本 compile.c 同之前类似，不单独贴出了，编译后生成二进制文件是 prog_pipe。下面是把 prog_pipe 写入根目录的代码，用过后要注释掉，见代码 15-45。

代码 15-45 （project/c15/j/kernel/main.c）

```
…略
21 int main(void) {
22      put_str("I am kernel\n");
23      init_all();
24
25 /*************    写入应用程序    *************/
26      uint32_t file_size = 5343;
27      uint32_t sec_cnt = DIV_ROUND_UP(file_size, 512);
28      struct disk* sda = &channels[0].devices[0];
29      void* prog_buf = sys_malloc(file_size);
30      ide_read(sda, 300, prog_buf, sec_cnt);
31      int32_t fd = sys_open("/prog_pipe", O_CREAT|O_RDWR);
32      if (fd != -1) {
33           if(sys_write(fd, prog_buf, file_size) == -1) {
34               printk("file write error!\n");
35               while(1);
36           }
37      }
38 /*************    写入应用程序结束    *************/
39     cls_screen();
40     console_put_str("[rabbit@localhost /]$ ");
41     thread_exit(running_thread(), true);
42     return 0;
43 }
…略
```

下面是运行的结果，如图 15-22 所示。

如图 15-22 所示，先执行 ls –l 查看文件写入的结果，prog_pipe 已经写入成功了，执行该命令后，prog_pipe 父进程先执行，第一行输出了“I`m father, my pid is 2”，然后此时父进程就退出了，需要其父进程 my_shell 为其善后。接着第二行输出的“child_pid 2，it`s status:8”是由 my_shell 获取其子进程 prog_pipe 后输出的，该子进程就是 prog_pipe 父进程。第三行输出的命令提示符“[rabbit@localhost /]”是 my_shell 捕获 prog_pipe 父进

程后的下一轮循环输出的。第四行是 prog_pipe 子进程运行，输出“I’m child，my pid is 5”，接着第五行输出父进程对自己所表达的内容。prog_pipe 父进程提前退出了，它在退出时，已经将其子进程过继给 init，因此 prog_pipe 子进程执行完后，为其做善后工作的是 init，此时 init 输出“I’m init，My pid is 1，I receive…”。

```
Bochs x86 emulator, http://bochs.sourceforge.net/
[rabbit@localhost /]$ ls -l
total: 192
d  0  192  .
d  0  192  ..
-  1  24  file1
d  2  96  dir1
-  5  4777  prog_no_arg
-  6  5307  prog_arg
-  7  5476  cat
-  9  5343  prog_pipe
[rabbit@localhost /]$ ./prog_pipe

I`m father, my pid is 2
child_pid 2, it's status: 8
[rabbit@localhost /]$
I`m child, my pid is 5
I`m child, my father said to me: "Hi, my son, I love you!"
I`m init, My pid is 1, I recieve a child, It`s pid is 5, status is 9

[rabbit@localhost /]$ rm cat
[rabbit@localhost /]$ rm prog_arg
[rabbit@localhost /]$ rm prog_no_arg
[rabbit@localhost /]$ ls
. .. file1 dir1 prog_pipe
[rabbit@localhost /]$
IPS: 18.891M  NUM CAPS SCRL HD:0-M HD:0-S
```

▲图 15-22　父子进程间通过管道通信

最后执行了三个 rm 命令，把 cat、prog_arg 和 prog_no_arg 这三个测试程序都删除了。

目测运行结果是正确的，因此本节到这就结束了，咱们还有最后一节。

15.7.5　在 shell 中支持管道

今天我们让 shell 支持管道操作。

管道操作大伙儿都了解吧，很多命令行界面都支持此类操作，比如 Windows 命令行窗口和 Linux 的 shell，管道符是‘|’，在命令行中可以有多个管道符，在管道符的左右两端各有一条命令，因此命令行中若包含管道符，至少要有两条命令。在命令行中支持管道通常是为了数据的二次加工、过滤出感兴趣的部分，比如“ps -ef|grep php-cgi”，这样会把 php-cgi 的信息从进程列表中过滤出来，但这样输出的信息中又包括 grep 命令本身，因此一般用双层管道：“ps -ef|grep php-cgi|grep –v grep”，其中“grep –v grep”是过滤出不包含 grep 的文本行，这样输出的信息就全是 php-cgi 的信息。

管道之所以可以这样用，原因是利用了输入输出重定向。通常情况下键盘是程序的输入，屏幕是程序的输出，它们都是标准的输入输出，即之前所说的 stdin 和 stdout。既然有“标准的”输入输出，就一定存在非标准的情况，这就是输入输出重定向。如果命令的输入并不来自于键盘，而是来自于文件，这就称为输入重定向，如果命令的输出并不是屏幕，而是想写入到文件，这就称为输出重定向。利用输入输出重定向的原理，可以将一个命令的输出作为另一个命令的输入。因此命令行中若包括管道符，则将管道符左边命令的输出作为管道符右边命令的输入。

管道操作的原理就是这样，以上所说的似乎和平时了解的差不多，如果觉得依然只是在表面上陈述，并没有说到骨子里，除了我个人表述的原因外，估计就是缺乏实践经验造成的，任何知识在缺乏实际操作经验的情况下都显得“虚无缥缈、飘忽不定”，因此只能在实际代码中理解了。

管道的核心就是输入输出重定向，称为核心其实实现起来并不难，再加上咱们本身的定位就是入门……不啰嗦了，见代码 15-46。

代码 15-46　（project/c15/k/shell/pipe.c）

```
…略
37 /* 将文件描述符old_local_fd重定向为new_local_fd */
38 void sys_fd_redirect(uint32_t old_local_fd, uint32_t new_local_fd) {
39     struct task_struct* cur = running_thread();
```

```
40     /* 针对恢复标准描述符 */
41     if (new_local_fd < 3) {
42         cur->fd_table[old_local_fd] = new_local_fd;
43     } else {
44         uint32_t new_global_fd = cur->fd_table[new_local_fd];
45         cur->fd_table[old_local_fd] = new_global_fd;
46     }
47 }
…略
```

函数 sys_fd_redirect 接受 2 个参数，旧文件描述符 old_local_fd、新文件描述符 new_local_fd，功能是将文件描述符 old_local_fd 重定向为 new_local_fd。

函数原理很简单，我们知道文件描述符是 pcb 中数组 fd_table 的下标，数组元素的值是全局文件表 file_table 的下标，因此很容易想到，文件描述符重定向的原理就是：将数组 fd_table 中下标为 old_local_fd 的元素的值用下标为 new_local_fd 的元素的值替换。

另外，pcb 中文件描述符表 fd_table 和全局文件表 file_table 中的前 3 个元素都是预留的，它们分别作为标准输入、标准输出和标准错误（未实现，但依然预留），因此，如果 new_local_fd 小于 3 的话，不需要从 fd_table 中获取元素值，可以直接把 new_local_fd 赋值给 fd_table[old_local_fd]，而这通常用于将输入输出恢复为标准的输入输出，下面看实现。

第 39 行获取了当前线程 cur，第 41～42 行对标准输入输出做了特殊处理，如果 new_local_fd 小于 3，直接将 new_local_fd 给 cur->fd_table[old_local_fd] 赋值，否则，第 44～45 行，先获得 new_local_fd 对应的 file_table 下标 new_global_fd，然后将 new_global_fd 赋值给 cur->fd_table[old_local_fd]，至此完成了重定向。

下面还要在 shell.c 中增加代码，见代码 15-47。

代码 15-47 （project/c15/k/shell/shell.c）

```
…略
117 /* 执行命令 */
118 static void cmd_execute(uint32_t argc, char** argv) {
119     if (!strcmp("ls", argv[0])) {
120          buildin_ls(argc, argv);
121     } else if (!strcmp("cd", argv[0])) {
122          if (buildin_cd(argc, argv) != NULL) {
123               memset(cwd_cache, 0, MAX_PATH_LEN);
124               strcpy(cwd_cache, final_path);
125          }
126     } else if (!strcmp("pwd", argv[0])) {
…略
166 char* argv[MAX_ARG_NR] = {NULL};
167 int32_t argc = -1;
168 /* 简单的 shell */
169 void my_shell(void) {
170     cwd_cache[0] = '/';
171     while (1) {
172         print_prompt();
173         memset(final_path, 0, MAX_PATH_LEN);
174         memset(cmd_line, 0, MAX_PATH_LEN);
175         readline(cmd_line, MAX_PATH_LEN);
176         if (cmd_line[0] == 0) {    // 若只键入了一个回车
177             continue;
178         }
179
180         /* 针对管道的处理 */
181         char* pipe_symbol = strchr(cmd_line, '|');
182         if (pipe_symbol) {
183     /* 支持多重管道操作，如 cmd1|cmd2|..|cmdn,
184      * cmd1 的标准输出和 cmdn 的标准输入需要单独处理 */
185
186     /*1 生成管道*/
187          int32_t fd[2] = {-1};      // fd[0]用于输入，fd[1]用于输出
188           pipe(fd);
189          /* 将标准输出重定向到 fd[1],
           使后面的输出信息重定向到内核环形缓冲区 */
190          fd_redirect(1,fd[1]);
191
```

```
192     /*2 第一个命令 */
193          char* each_cmd = cmd_line;
194          pipe_symbol = strchr(each_cmd, '|');
195          *pipe_symbol = 0;
196
197          /* 执行第一个命令，命令的输出会写入环形缓冲区 */
198          argc = -1;
199          argc = cmd_parse(each_cmd, argv, ' ');
200          cmd_execute(argc, argv);
201
202          /* 跨过'|'，处理下一个命令 */
203          each_cmd = pipe_symbol + 1;
204
205          /* 将标准输入重定向到 fd[0]，使之指向内核环形缓冲区*/
206          fd_redirect(0, fd[0]);
207     /*3 中间的命令,命令的输入和输出都是指向环形缓冲区 */
208          while ((pipe_symbol = strchr(each_cmd, '|'))) {
209              *pipe_symbol = 0;
210              argc = -1;
211              argc = cmd_parse(each_cmd, argv, ' ');
212              cmd_execute(argc, argv);
213              each_cmd = pipe_symbol + 1;
214           }
215
216     /*4 处理管道中最后一个命令 */
217           /* 将标准输出恢复屏幕 */
218           fd_redirect(1,1);
219
220          /* 执行最后一个命令 */
221           argc = -1;
222           argc = cmd_parse(each_cmd, argv, ' ');
223           cmd_execute(argc, argv);
224
225     /*5  将标准输入恢复为键盘 */
226           fd_redirect(0,0);
227
228     /*6 关闭管道 */
229           close(fd[0]);
230           close(fd[1]);
231       } else {         // 一般无管道操作的命令
232           argc = -1;
233           argc = cmd_parse(cmd_line, argv, ' ');
234           if (argc == -1) {
235                printf("num of arguments exceed %d\n", MAX_ARG_NR);
236                continue;
237           }
238           cmd_execute(argc, argv);
239        }
240     }
241     panic("my_shell: should not be here");
242 }
```

本节中把 shell.c 中原本判断内建、外部命令的一堆 if else 封装到第 118 行的函数 cmd_execute 中，不多说了，本次对管道的处理是函数 my_shell 中第 181～230 行。

第 181 行通过 strchr 函数在 cmd_line 中寻找管道字符'|'，如果找到，pipe_symbol 的值则为字符'|'的地址，下面讨论下处理管道命令的思路。

在命令行中可以出现多个管道符接连过滤数据的情况，比如“cmd1|cmd2|..|cmdn”，这其中包括了 n 个命令的接力配合，我们称之为多重管道操作。我们讨论过了，管道操作中前一个命令的输出作为后一个命令的输入，cmd1 是第 1 个命令，没人为它提供输入，因此其输入不变，仍为标准输入，但其输出是要传给命令 cmd2，因此 cmd1 的标准输出不能指向屏幕了，必须要重定向到管道的环形缓冲区中，命令 cmd2 的标准输入必须也重定向到管道的环形缓冲区才能够获得 cmd1 的输出，cmd2 的输出为了传给 cmd3，必须也要将标准输出重定向到管道环形缓冲区，cmd4 为了获得 cmd3 的输出结果，必须将 cmd4 的标准输入重定向到管道环形缓冲区……依次类推，当执行到命令 cmdn 时，cmdn 的标准输入必须要指向管道环形缓冲区才能获得命令 cmdn-1 提供的输出，但 cmdn 是最后一个命令，它要将结果打印到屏幕，因此其标

准输出不用改变，依然为屏幕。也就是说，除 cmd1 的标准输入和 cmdn 的标准输出不变外，其他命令的标准输入和输出都要重定向到管道。下面分六步来完成管道操作。

第 186～191 行完成第一步，生成管道，这是调用 pipe 系统调用完成的。第 190 行调用 fd_redirect(1,fd[1]) 将标准输出重定向到用于写管道的文件描述符 fd[1]，至此程序的输出都写到管道中。

第 193～206 行开始第二步，解析第 1 个命令并执行。命令行中的各个命令是用指针 each_cmd 记录的，它指向各命令在 cmd_line 中的地址。解析出命令后调用 cmd_execute 执行，然后在第 203 行使 pipe_symbol 加 1，跨过 cmd_line 中的相应的'|'。在执行第 2 个命令之前，在第 206 行执行“fd_redirect(0,fd[0])”将标准输入重定向到管道，这样第 2 个命令才能获得第 1 个命令的输出。

第 208～214 行完成第三步，循环处理 cmd2～cmdn-1，此时它们的标准输入和输出都已指向管道，继续解析命令并执行就可以了，不多说了。

执行完 while 循环后，第 218～223 行完成第四步，调用“fd_redirect(1,1)”将标准输出恢复为屏幕，然后第 223 行执行最后一个命令，此时命令的输出信息会在屏幕上显示。

第 226 行是第五步，调用“fd_redirect(0,0)”将标准输入恢复为键盘。

第 229～230 行是第六步，将管道关闭。至此管道的处理就完成了。第 231～239 行是一般无管道符的处理。

按理说该是测试的时候了，可我们还没有从标准输入获取数据的用户程序呢，立即把之前的 cat.c 改改，使 cat 无参数时，默认从键盘获取数据，见代码 15-48。

代码 15-48 （project/c15/k/command/cat.c）

```
 1 #include "syscall.h"
 2 #include "stdio.h"
 3 #include "string.h"
 4 int main(int argc, char** argv) {
 5     if (argc > 2) {
 6         printf("cat: argument error\n");
 7         exit(-2);
 8     }
 9
10     if (argc == 1) {
11         char buf[512] = {0};
12         read(0, buf, 512);
13         printf("%s",buf);
14         exit(0);
15     }
16
17     int buf_size = 1024;
18     char abs_path[512] = {0};
19     void* buf = malloc(buf_size);
20     if (buf == NULL) {
21         printf("cat: malloc memory failed\n");
22         return -1;
23     }
24     if (argv[1][0] != '/') {
25         getcwd(abs_path, 512);
26         strcat(abs_path, "/");
27         strcat(abs_path, argv[1]);
28     } else {
29         strcpy(abs_path, argv[1]);
30     }
31     int fd = open(abs_path, O_RDONLY);
32     if (fd == -1) {
33          printf("cat: open: open %s failed\n", argv[1]);
34          return -1;
35     }
36     int read_bytes= 0;
37     while (1) {
38        read_bytes = read(fd, buf, buf_size);
39        if (read_bytes == -1) {
40             break;
41        }
42        write(1, buf, read_bytes);
```

```
43     }
44     free(buf);
45     close(fd);
46     return 66;
47 }
```

这个版本的 cat.c 就是在上一版的基础上，加了第 10～15 行，当无参数时，直接调用 read 系统调用从键盘获取数据。编译还是用 compile.sh 就行了，同之前类似，不贴代码了。

下面是在 main.c 中将 cat 写入。另外说一下，在上节的图 15-22 中，我们已经将根目录下曾经的测试用例删除了，目录根目录中只有普通文件 file1，目录 dir1，程序 prog_pipe 以及“.”和“..”。所以下面代码中可以在根目录中写入新的 cat 程序，见代码 15-49。

代码 15-49　（project/c15/k/kernel/main.c）

```
…略
21 int main(void) {
22     put_str("I am kernel\n");
23     init_all();
24
25 /*************    写入应用程序    *************/
26     uint32_t file_size = 5698;
27     uint32_t sec_cnt = DIV_ROUND_UP(file_size, 512);
28     struct disk* sda = &channels[0].devices[0];
29     void* prog_buf = sys_malloc(file_size);
30     ide_read(sda, 300, prog_buf, sec_cnt);
31     int32_t fd = sys_open("/cat", O_CREAT|O_RDWR);
32     if (fd != -1) {
33          if(sys_write(fd, prog_buf, file_size) == -1) {
34               printk("file write error!\n");
35               while(1);
36          }
37     }
38 /*************    写入应用程序结束   *************/
39     cls_screen();
40     console_put_str("[rabbit@localhost /]$ ");
41     thread_exit(running_thread(), true);
42     return 0;
43 }
…略
```

另外，为了显示系统支持的命令，我加了个内建命令 help，当在 shell 中输入 help 时，系统会打印支持的内建命令及快捷键。原理是实现了 help 系统调用，下面是 help 对应的 sys_help 代码，它定义到了 fs.c 中，见代码 15-50。

代码 15-50　（project/c15/k/fs/fs.c）

```
…略
891 /* 显示系统支持的内部命令 */
892 void sys_help(void) {
893     printk("\
894  buildin commands:\n\
895        ls: show directory or file information\n\
896        cd: change current work directory\n\
897        mkdir: create a directory\n\
898        rmdir: remove a empty directory\n\
899        rm: remove a regular file\n\
900        pwd: show current work directory\n\
901        ps: show process information\n\
902        clear: clear screen\n\
903  shortcut key:\n\
904        ctrl+l: clear screen\n\
905        ctrl+u: clear input\n\n");
906 }
…略
```

添加系统调用的过程就不多说了，图 15-23 所示是目前所有的系统调用号。

编译运行之后，下面的两张图是运行结果，如图 15-24 和图 15-25 所示。

```
7 enum SYSCALL_NR {
8     SYS_GETPID,
9     SYS_WRITE,
10    SYS_MALLOC,
11    SYS_FREE,
12    SYS_FORK,
13    SYS_READ,
14    SYS_PUTCHAR,
15    SYS_CLEAR,
16    SYS_GETCWD,
17    SYS_OPEN,
18    SYS_CLOSE,
19    SYS_LSEEK,
20    SYS_UNLINK,
21    SYS_MKDIR,
22    SYS_OPENDIR,
23    SYS_CLOSEDIR,
24    SYS_CHDIR,
25    SYS_RMDIR,
26    SYS_READDIR,
27    SYS_REWINDDIR,
28    SYS_STAT,
29    SYS_PS,
30    SYS_EXECV,
31    SYS_EXIT,
32    SYS_WAIT,
33    SYS_PIPE,
34    SYS_FD_REDIRECT,
35    SYS_HELP
36 };
```

▲图 15-23　系统调用号

```
Bochs x86 emulator, http://bochs.sourceforge.net/
[rabbit@localhost /]$ cd dir1
[rabbit@localhost /dir1]$ ls -l|../cat
total: 96
d  2  96   .
d  0  144  ..
d  3  72  dir11
-  8  910  cat.c
child_pid 2, it's status: 0
[rabbit@localhost /dir1]$ ls -l|../cat|/cat|../cat
total: 96
d  2  96   .
d  0  144  ..
d  3  72  dir11
-  8  910  cat.c
child_pid 2, it's status: 0
child_pid 2, it's status: 0
child_pid 2, it's status: 0
[rabbit@localhost /dir1]$ ps
PID            PPID           STAT           TICKS          COMMAND
1              NULL           WAITING        54             init
3              NULL           BLOCKED        5AB9D0         idle
4              1              RUNNING        3E1            init
[rabbit@localhost /dir1]$ _
IPS: 24.247M
```

▲图 15-24　管道处理

```
Bochs x86 emulator, http://bochs.sourceforge.net/
[rabbit@localhost /]$ help
 buildin commands:
       ls: show directory or file information
       cd: change current work directory
       mkdir: create a directory
       rmdir: remove a empty directory
       rm: remove a regular file
       pwd: show current work directory
       ps: show process information
       clear: clear screen
 shortcut key:
       ctrl+l: clear screen
       ctrl+u: clear input

[rabbit@localhost /]$ _
IPS: 22.363M
```

▲图 15-25　系统帮助

目测符合预期，因此本节就到这了。

参考文献

1.《x86/x64 体系探索及编程》，作者邓志。
2.《深入理解 Linux 内核（第 3 版）（涵盖 2.6 版）》，作者博韦等。
3.《Linux 内核设计与实现（原书第 3 版）》，作者拉芙（Robert Love），译者陈莉君、康华。
4.《深入 Linux 内核架构》，作者莫尔勒（Wolfgang Mauerer），译者郭旭。
5.《Linux 内核完全剖析》，作者赵炯。
6.《Linux 内核源代码情景分析（上、下册）》，作者毛德操、胡希明。
7.《一个操作系统的实现》，作者于渊。
8.《自己动手写嵌入式操作系统》，作者蓝枫叶。
9.《操作系统设计与实现（第三版）（上、下册）》，作者安德鲁（Andrew S.Tanenbaum）、塔嫩鲍姆（Albert S.Woodhull）。
10.《现代操作系统（原书第 3 版）》，作者塔嫩鲍姆（Tanenbaum.A.S），译者陈向群、马洪兵。
11.《计算机的心智操作系统之哲学原理（第 2 版）》，作者邹恒明。
12.《C 语言入门经典（第 5 版）》，作者霍尔顿（Ivor Horton）。
13.《C 语言程序设计：现代方法（第 2 版）》，作者金（K.N.King）。
14.《Linux 程序设计（第 4 版）》，作者马修（Neil Matthew）、斯通斯（Richard Stones）。
15. 斯坦福大学教学操作系统 Pintos。
16.《x86 汇编语言：从实模式到保护模式》，作者李忠、王晓波。
17.《汇编语言（第 3 版）》，作者王爽。
18.《基于 Linux 系统的汇编语言程序设计》，作者程楠。
19.《Linux 内核完全注释》，作者赵炯。
20.《微机原理与接口技术教程》，作者王克义、鲁守智。
21.《微机原理与接口技术（第 2 版）》，作者周明德、蒋本珊。
22.《INTEL 开发手册三卷》。
23.《跟我一起写 makefile》。
24.《GNU gcc 嵌入式系统开发》，作者董文军。

以上列举不详尽，如有遗漏请见谅。


笔记栏




笔记栏



异步社区
人民邮电出版社
www.epubit.com.cn

欢迎来到异步社区！

异步社区的来历

异步社区（www.epubit.com.cn）是人民邮电出版社旗下 IT 专业图书旗舰社区，于 2015 年 8 月上线运营。

异步社区依托于人民邮电出版社 20 余年的 IT 专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与 POD 按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。



社区里都有什么？

购买图书

我们出版的图书涵盖主流 IT 技术，在编程语言、Web 技术、数据科学等领域有众多经典畅销图书。社区现已上线图书 1000 余种，电子书 400 多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100 积分 =1 元，购买图书时，在 0 使用积分 里填入可使用的积分数值，即可扣减相应金额。

特 别 优 惠

购买本书的读者专享**异步社区购书优惠券**。

使用方法：注册成为社区用户，在下单购书时输入 57AWG 使用优惠码，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于 Markdown 的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握 IT 圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：

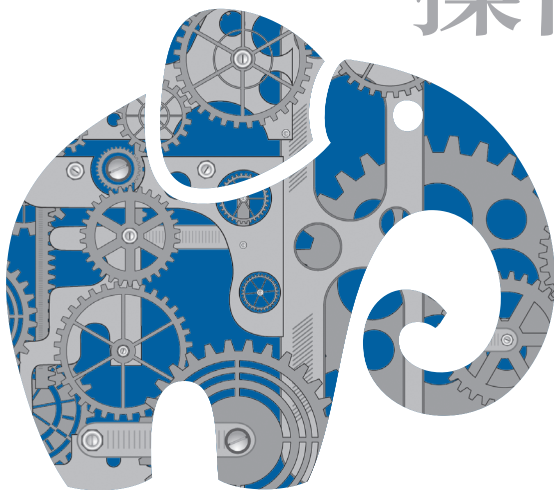
				
异步社区	微信服务号	微信订阅号	官方微博	QQ 群：368449889

社区网址：www.epubit.com.cn

官方微信：异步社区

官方微博：@人邮异步社区，@人民邮电出版社 - 信息技术分社

投稿 & 咨询：contact@epubit.com.cn



操作系统真象还原

一本看得懂，学得会，深入理解操作系统原理的原创精品书

在开源的世界里，最耀眼的操作系统莫过于 Linux。不过，如果您想以短、平、快的方式了解操作系统的实现原理，阅读 Linux 源码并不是一个好的选择，其代码量非常大，已经到了上千万行的级别。

Linux 中 90% 以上的代码都是用在资源管理、策略、算法及数据结构等方面，这正是它优秀的地方，恰恰也是给初学者造成困扰的地方。

基于让读者快速掌握操作系统原理的想法，本书具有的特点如下：

用最少的代码展示操作系统的本质

本书用较少的代码实现了一个功能完备的操作系统，有效代码仅为 6023 行，从学习 Linux 的千万行降到研究几千行代码，大大降低了学习操作系统原理的门槛。

本书实现的操作系统的特点是程序量少，功能多

实现了内核线程、特权级变换、进程、任务调度、fork、exec、父子进程间的通信等。

用实际代码解释了锁、信号量及生产者消费者问题等操作系统中的难点，让这些深奥的技术易学习、易理解。支持文件系统、管道及 shell 操作等。

通俗易懂，易学易用

用通俗易懂、诙谐幽默的语言解释了操作系统的实现原理。

每节一个知识点，在实战中逐步实现一个完整的操作系统。

ISBN 978-7-115-41434-2



封面设计：董志桢

分类建议：计算机／操作系统开发
计算机／程序设计

人民邮电出版社网址：www.ptpress.com.cn